DEPARTMENT OF DEFENCE
DEFENCE SCIENCE & TECHNOLOGY ORGANISATION
DSTO

Translating Deeply Structured Information

Darryn J. Reid

DSTO-TR-0936

Translating Deeply Structured Information

Darryn J Reid

Information Technology Division
Electronics and Surveillance Research Laboratory

DSTO-TR-0936

ABSTRACT

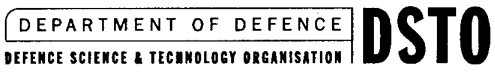
The problem of interfacing Command, Control, Communications and Intelligence (C3I) systems and applicable simulations is considered. In particular, this document focusses on the requirements and design of an engine to support translation between systems expecting to communicate using dissimilar message languages.

This engine interprets a given behavioural specification written in a high-level declarative language built around standard SQL. It is therefore natural and convenient to consider implementation using an appropriate relational database system to facilitate data storage and manipulation.

Beginning with an overview of the broader context and background, the discussion considers the way in which the semantic structures of the input and output languages can be captured using a conceptual modeling language. Such models can be readily mapped into a relational schema, and the actions that a translator should perform are easily expressed using SQL. Each of these actions must occur when a given set of conditions is satisfied; the engine is therefore a specialised rule-based system that manipulates the tuples of a relational database.

RELEASE LIMITATION

Approved for public release



Published by

DSTO Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury South Australia 5108 Australia

Telephone: (08) 8259 5555
Fax: (08) 8259 6567

AR-011-201
February 2000

APPROVED FOR PUBLIC RELEASE

Translating Deeply Structured Information

Executive Summary

Creating a virtual environment often involves constructing interfaces between systems that were never designed to interact. These same fundamental barriers to interoperability are observed whenever independently constructed information systems are brought together to address some new organisational need. The focus here is on systems that expect to communicate in discrete chunks of structured information such as paragraphs or formatted messages.

At the core of any interface between a pair of such systems is the ability to manipulate incoming information to generate appropriate structures that can be expressed in the output language. The required behaviour of a translator to perform this function can be derived from a suitable definition of the semantics of the input and output languages. Object-Role Modelling (ORM) is used in this document to express the semantic structures of either language; however, other suitable conceptual modelling paradigms such as Entity-Relationship (E-R) diagrams could be employed instead.

The specification of the translator is most conveniently expressed as a set of rules, each relating the information held by the translator at any moment to some actions that should be subsequently performed. Such preconditions and actions can be readily expressed using standard SQL; this is a direct result of the fact that models built using ORM (or E-R diagrams) are readily mapped into relational database schemata. The focus of this document is the design of a translator engine capable of interpreting any set of such rules.

The rapid development of reliable interfaces to facilitate the interoperability of systems that utilise different message languages can be achieved by employing a systematic and rigorous approach to defining the exact nature of the interactions. A flexible and reusable system for supporting this activity is proposed, in the form of a translator engine to interpret a given behavioural specification at run-time.

Author

Darryn J Reid
Information Technology Division

Darryn J Reid completed the BSc and BscHons degrees in mathematics and computer science from the University of Queensland in 1991 and 1992 respectively, and received a PhD from the University of Queensland in 1995. Darryn Reid joined DSTO as a Senior Professional Officer Grade C in Information Technology Division in 1995. In 1996 he became a Research Scientist in Information Technology Division, and joined Electronic Warfare Division as a Senior Research Scientist in 1999.

Contents

1. Introduction and Overview

Consider the problem of constructing an interface between two independently constructed systems that represent similar concepts, but communicate information about these concepts in different forms [22]. Building a parser to decompose incoming information and placing it in internal data structures is relatively easy; numerous parser generator tools are available to assist in this task [1,2,7]. Likewise, composing an output message from data held internally is really a programming exercise. While this is not necessarily trivial to implement, this document focuses instead on how the information in the incoming message stream can be manipulated to produce information that can be used to build consistent and correct output messages. The component of the interface that achieves this will be termed a 'translator', to emphasise that it processes information according to its meaning, rather than just manipulating pieces of data.

Imagine now that such a translator is to be implemented. How should this problem be approached, and what kinds of operations will the translator need to perform? More precisely, what methodology should be employed to specify how the translator should operate? Can it be guaranteed that the meaning of information will be preserved in the translation? What resource demands will such a project incur, in terms of the programming effort required to realise an implementation? And, ultimately, what support can be developed to alleviate the burden of implementing a new solution each time the need to translate between different message streams is encountered?

To begin to develop some answers to these and other questions, suppose that a translator to accept a stream of messages of one language and produce another stream conforming to a different language is to be designed and programmed. Specifying the behaviour of the translator demands some analysis of the two languages and relationships between them [3,4,8-15,21,23-25]. That is, the meaning of the information contained in all the messages of both languages must be expressed in some appropriate manner; this is precisely what the conceptual modelling languages used in information system design are intended to do [16,20].

One such way of capturing this kind of structural information is to use Object-Role Modelling (ORM) language [18] for database conceptual schemata; this is an intuitively appealing graphical method of expressing relationships between various domains of data values. The process by which an ORM diagram is obtained will not be discussed in detail here, other than to say that the result is a single model that describes the semantic structure of information in the message types of both input and output languages. Further information regarding schema design and issues in schema integration is available elsewhere [3,8,10,12,14,15,17,19,21,23,24].

As an aside, most of the examples presented in this document operate implicitly within the closed-world assumption, meaning that it is presumed that all relevant facts are represented within the system. Addressing such concerns in database systems, knowledge base systems, logic programming languages and automated reasoning is still an active area of research [17], and no complete solution is currently known.

Figure 1 below is an example ORM diagram that represents some of the information about targets that both the ADFORMS and OTH-T Gold message formats used in military C3I systems and simulations [5,6,26] can convey.

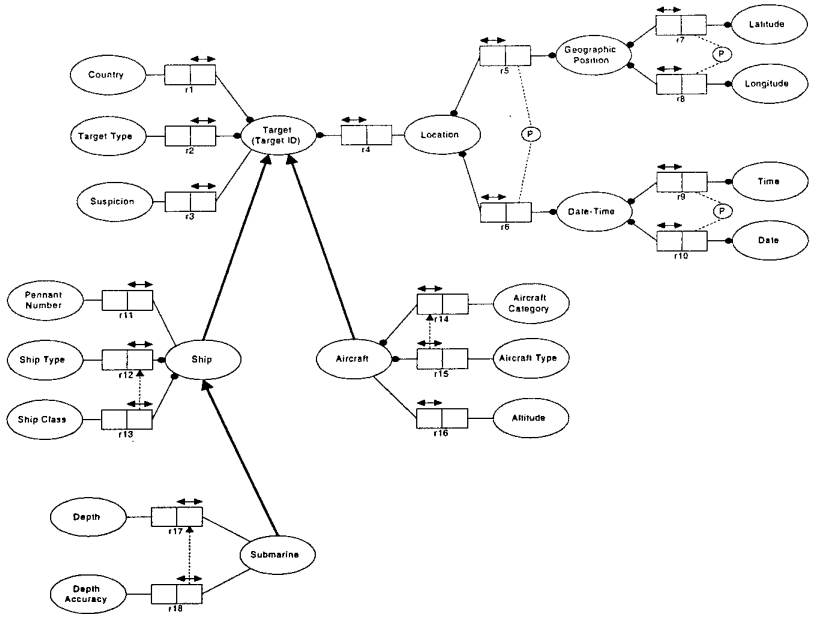


Figure 1: *An example ORM schema.*

Every message type of either format is described by a subset of the relationships expressed on the ORM diagram. Given all of this information, designing the translator amounts to finding sequences of operations that will transform sets of input messages into sets of output messages. Each of these operations is an inference operation that recombines some of the relationships defined on the schema to produce a new derived relationship, and corresponds to an SQL query on a database that would implement the conceptual schema. For example, suppose that it is known that certain countries operate some classes of ships (Australia and New Zealand have ANZAC Frigates), as shown in the schema fragment of Figure 2. Notice that there are no mandatory role constraints, because this information may not be stored for all ship classes or all countries.

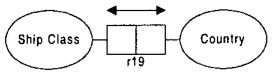


Figure 2: *A schema fragment.*

Now consider an input message containing information about some ship, without indicating its country of origin. The output message format, however, demands that a

country (which may include the value 'unknown') be given for all targets. If it occurs that the ship in question is of a class that only one country operates (for example, only Australia possesses Collins class submarines), then the country can be inferred. An SQL statement to determine the possible countries of origin might be

```
SELECT Country FROM ClassOperators WHERE ShipClass = 'Collins'.
```

If this returns a relation containing a single value, then that value will be used as the country code in the output message. If more than one value is returned, then the country cannot be determined from the ship's class alone.

Some sequences of derivations may be obvious, while others may be more difficult to ascertain. Imagine that the input language does not explicitly indicate a suspicion code, which is an indication of what kind of mission the target appears to be undertaking. On the other hand, the output message format explicitly requires this information to be provided. It is conceivable that the suspicion code for an aircraft, say, can be inferred by a sequence of operations that relate the aircraft type, country of origin, position and altitude, and the locations of other targets. Such inferences are not limited to being deterministic; if the conceptual modelling language were a Bayesian belief network, for instance, the determination of the suspicion code would be stochastic. Indeed, it is possible to bury fuzzy and stochastic relationships within the ORM schema: some of its relationships may represent complicated computations rather than explicitly stored tuples.

The process of finding what operations should be performed is made even more complex by the possible temporal separation of different pieces of information. That is, information related with respect to the sequences of operations mentioned earlier could be spread across several messages (of either or both of the input and output languages).

To see this, consider the various ways in which the messages of the incoming stream may map into output messages. Firstly, it is plausible that some kind of received message might be best mapped into a single output message. This is the simplest possibility, and the easiest to implement, in which all of the information in the input message that can be expressed in the output language is contained within the single outgoing message.

Alternatively, the input message might be broken into a corresponding set of several output messages. This means that all of the information in the incoming message that can be expressed in the output language is spread across a number of message types of the output language. An example of this occurs in considering translating from OTH-T Gold to ADFORMS: some forms of the Gold Contact Report message contain information that ADFORMS disperses across several messages, including the Location Report and Hostile Air Contact Report messages, among others.

Another possibility is the reverse of the second: information needed to build an output message lies spread across several input messages. This is more difficult to implement, as it requires information from incoming messages to be stored until all of the information that is required by the output message is available. One situation in which

this occurs is the reverse of the example given above, namely in translating from ADFORMS to OTH-T Gold.

The general conclusion that can be drawn from all of these examples is that a system for generating a translator must be able to consider ways of mapping any number m of input messages into n messages of the output stream. In implementing this, the translator must be capable of storing information for later use, without knowing if or when the other required input messages will arrive. To make matters worse, it is conceivable that more than one message type in the input language might provide some of the information needed to produce the output.

The outcome of these observations is that the process of generating a translator should focus on the current state of available knowledge at any moment during the translator's operation. In this way, a strategy can be devised that determines the next action to take, in order that some measure of long-term performance is maximised. This performance measure will describe the overall quality of the result of the translator, in terms of the amount of information that is successfully delivered as output messages.

Certainly, this implies the existence of a trade-off between the somewhat competing requirements of rapid delivery of information and the completeness of that information. Note that the translator engine itself is fully capable of supporting the full range of decisions that might be contemplated, but in this discussion the focus will be generally on providing output messages that are as complete as possible.

A further complication that can be observed in message formats such as OTH-T Gold and ADFORMS is the overlap between different message types. That is to say, several message types of either format can contain some of the same information. As a result, the mapping from input stream to output stream is not unique, although certain possible solutions might clearly be more desirable than others.

To support the process of constructing translators from a definition of the structural relationships between the various data domains, some automated translator development tool is required. The ultimate goal is to reduce or eliminate the need to hard-code a solution every time an ability to convert between some pair of languages is required.

2. First Steps Towards a Solution

In recognition of the complexity of the problem, a formulation as a sequential decision process has been developed to support the automated generation of a translator from the schema-level description of the relationships between the input and output languages. The behaviour of the translator is mathematically described in terms of sequences of actions each of which moves the system from one state of knowledge to another. In constructing a translator, the objective is to discover such sequences for which a measure of performance is maximal.

It is fundamentally important to note that the operations that the on-line translator will perform, and the algorithms used by the translator generator to specify these actions, are ignorant of the actual data being processed. That is to say, the behaviour of the translator is completely specified in terms of the conceptual schema representation of the interface, not in terms of any individual data values that may arise during the translator's operation.

The translator maintains a database in which information is stored and manipulated. This database is divided into two sections, one to represent the combined schema (or the common subschema), and the another to contain derived relationships. The structure of the first part of the database is fixed, although its data content might vary. On the other hand, the derived relationships will be produced and discarded as the result of actions taken by the translator.

The fixed component of the database will typically be instantiated with data specific to the particular circumstances in which the translator is to operate. For example, some scenario might involve certain vessels and aircraft, so the translator could be initialised with potentially useful information about these specific entities. Some predefined information of this variety that might be made available to the translator is shown in Figure 3. Columns for which the title is underlined are keys, meaning that any value can appear in that column at most once. In this case, either pennant number or name uniquely identifies the particular vessel.

Pennant Number	Ship Name	Ship Type	Ship Class
02	Canberra	FFG07	FFG
74	Farncomb	Collins	SSK
39	Hobart	Charles F	DDG
73	Collins	Collins	SSK
62	Otama	Oberon	SSK
05	Melbourne	FFG07	FFG

Aircraft Type	Aircraft
F/A-18	F
F/A-18	A
Su-27M	B
MiG-29	F
F-111	B

Figure 3: *Some example predefined scenario data.*

This represents a collection of relevant facts that are known prior to the actual execution of the translator, and believed to be correct and complete. It is conceivable that additional information of this kind could be collected while in operation and stored for later reference; notice that it is the overall structure of this part of the database that remains fixed, while its contents may be permitted to change.

The variable part of the database holds information temporarily while it is being processed. As such, it is the schema-level structure of this component that defines the current state of knowledge as perceived by the search algorithm. Because information

from previous messages is encapsulated in the current state, the sequence of operations to perform does not explicitly depend on the message received, but rather on what information is available at that moment. This also gives the translator the ability to remember unused information from messages received at an arbitrary time in the past.

The example in Figure 4 illustrates the kind of information that might be temporarily held in the variable part of the database, at some instant. Information detailing the location of two targets has arrived, and perhaps the ship types for each has been derived from the ship names initially provided in the message. With further processing, such as deriving the ship class from the ship type, or splitting the position fields into latitude and longitude, these tables might be deleted.

Target	Position	Time
VN603	2345S13245E	04-021234Z
VN603	2346S13250E	04-060213Z
VP113	2352S1342E	04-101341Z

Target	Ship Type	Country
VN603	FFG07	AUS
VP113	Collins	AUS

Figure 4: *Some example data in temporary storage.*

Upon receipt of a message, the parser injects the relevant information it contains into the database. The translator will then perform some sequence of actions to manipulate this information, sometimes resulting in the production of one or more output messages. At other times, it may not be possible to produce any output until further information is gathered from future incoming messages.

Information that is used in producing an output is removed from the variable part of the database (discarded or sometimes shifted to the fixed part), while leftover information that might be useful in constructing future messages is retained for later. Such a mechanism is capable of remembering past input messages and, under the control of the translator generator, planning for those that may occur in the future.

3. A Translator Engine

The mathematical description of the translator generator supports either off-line or on-line implementations. In an off-line architecture, the translator generator would use the schema-level description of the interface to generate some form of intermediate code, which would then be executed by a run-time engine. The more general on-line option combines the translator and the translator generator into a single unit that would be also capable of minimising its response time (or some other measure of computational performance) by learning about the costs of performing different sequences of operations to achieve the same outcome.

By separating the translator technology into a run-time engine and a build-time translator generator, a significant new capability can be achieved, at greatly reduced risk and with lower development demands. For instance, the engine can be constructed independently, and in its own right would represent an enormous reduction in the amount of code that would need to be written to realise a translator. That is, the programmer would need only to specify a fairly high-level account of the behaviour required of the translator, with the engine itself managing the details of how such operations are performed. Because of this, an engine to support the receipt, manipulation, and delivery of structured information will be the focus of the initial stages of implementation.

A central issue in constructing the run-time engine is deciding upon the kinds of manipulations that it should be empowered to perform. Rather than focussing on database theory, a number of examples are presented here to illustrate the fundamental ideas underpinning the proposed design.

Suppose that information is available that gives the pennant numbers observed for some targets (see Figure 5). Note here that a particular ship might be reported by more than one observer, which results in more than one target number being assigned to the ship (this does occur in OTH-T Gold and ADFORMS messages). Information is also available that relates pennant numbers (which are unique to each vessel in this example) to other information about the particular ship in question.

Target	Pennant
VN603	02
VP113	73
VM371	02
VW292	39

Pennant Number	Ship Name	Ship Type	Ship Class
02	Canberra	FFG07	FFG
74	Farncomb	Collins	SSK
39	Hobart	Charles F	DDG
73	Collins	Collins	SSK
62	Otama	Oberon	SSK
05	Melbourne	FFG07	FFG

Figure 5: *Some received information and predefined facts.*

From this, it is possible to infer the ship name, ship type, and ship class that should be associated with each target number, using the natural join operator of relational algebra. The corresponding SQL for performing this operation might be

```
SELECT Target, PennantNumber, ShipName, ShipType, ShipClass FROM Targets, Ships
    WHERE Targets.PennantNumber = Ships.PennantNumber.
```

The natural join matches the rows of the first table with those of the second for which the specified columns have the same value. This undertaking corresponds to a

transitive inference on the underlying functional dependencies (which are expressed in the ORM diagram in terms of fact types and uniqueness constraints). In this case, the result is the new table shown as Figure 6.

Target	Pennant Number	Ship Name	Ship Type	Ship Class
VN603	02	Canberra	FFG07	FFG
VP113	73	Collins	Collins	SSK
VM371	02	Canberra	FFG07	FFG
VW292	39	Hobart	Charles F	DDG

Figure 6: *The result of the natural join.*

Of course, it is possible that only a few columns from this table are of interest, in which case the relational projection operator can be used to restrict the result. The natural join and projection operators would typically be combined into a single SQL query:

```
SELECT Target, ShipType FROM Targets, Ships
    WHERE Targets.PennantNumber = Ships.PennantNumber.
```

This produces instead a table containing only the 'Target' and 'Ship Type' columns, supposing that this is the information needed in ultimately producing an appropriate output message. The resulting relational table is illustrated below.

Target	Ship Type
VN603	FFG07
VP113	Collins
VM371	FFG07
VW292	Charles F

Figure 7: *The result of projection and natural join.*

Generalising this, the join of several tables will form a single table by combining together related rows. The exact structure of the join is determined by the pattern of relationships between the columns of all tables named in the query; in turn, the conceptual schema specifies all such relationships between all the tables of the database. That is to say, the list of all possible join queries that may be executed on a given database is defined by the design of the database schema.

Briefly, virtually all database schemata are naturally acyclic (that is, nonrecursive) in structure, and therefore almost every meaningful query will also be acyclic. Of these, by far the most common query is the linear 'chain query', which follows some referential path through the schema. To illustrate this, suppose that in some scenario each ship has a different name, and this uniquely determines its pennant number. The pennant number defines the class to which the ship belongs, and each ship class belongs to a single type of vessel. The schema fragment describing this is shown below in Figure 8; the referential path is clearly visible. In this illustration, a pennant number also uniquely determines a ship's name, establishing a one-one correspondence between these entity types.

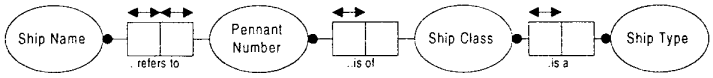


Figure 8: A referential path of length 3.

This structure would be realised as three tables in a relational database system (see Figure 9, below). The referential relationships between each of the tables are described in the relational data model as foreign keys; for instance, 'Ship Class' is said to be a foreign key in the second table that refers to the third table.

Ship Name	Pennant Number
Canberra	02
Farncomb	74
Hobart	39
Collins	73
Otama	62
Melbourne	05

Pennant Number	Ship Class
02	FFG07
74	Collins
39	Chales F
73	Collins
62	Oberon
05	FFG07

Ship Class	Ship Type
Oberon	SSK
FFG07	FFG
Collins	SSK
Charles F	DDG

***Figure 9:** Some tables of data corresponding to the schema of Figure 8.*

The natural join of these three tables is a chain query and would produce exactly the first table of Figure 3. To relate ship names only to ship types, this natural join operation followed by a projection onto the 'Ship Name' and 'Ship Type' attributes would be used. In SQL, the query to retrieve this result might be

```
SELECT ShipName, ShipType FROM Ships, ShipsClasses, ClassesTypes
    WHERE Ships.PennantNumber = ShipsClasses.PennantNumber
        AND ShipsClasses.ShipClass = ClassesTypes.ShipClass.
```

By far the most common activity of the translator will be realising inferences of this nature. Nonetheless, more complex manipulations will also be needed. For example, suppose that information is received that lists either the ship type or the aircraft type for each target. Predefined information about the scenario lists the countries that operate different ship types, and those that operate various kinds of aircraft. Note that the second and third tables have composite keys, meaning that only combinations of values in the underlined columns are unique. That is, a country can operate many different ship or aircraft types, and a type of ship or aircraft can potentially be operated by more than one country.

Target	Target Type
VN603	FFG07
VP337	MiG-29
VC717	F/A-18
VW292	Charles F
VA920	F-16

Ship Type	Country
Charles F	AUS
FFG07	AUS
Charles F	USA
Collins	AUS

Aircraft Type	Country
F-111	AUS
F/A-18	USA
F/A-18	AUS
F/A-18	MAL
F-16	USA
MiG-29	MAL

Figure 10: *An example for a more complicated inference.*

The following query might be used to list the targets, target types, and country of origin only for those targets for which the country can be unambiguously inferred. The first part retrieves naval targets for which the number of possible countries of origin is exactly 1, and the second part repeats the process for aircraft.

```
SELECT Target, TargetType, Country FROM Targets, ShipOperators
        WHERE TargetType = ShipType
        GROUP BY Target, Country HAVING COUNT (*) = 1
UNION
SELECT Target, TargetType, Country FROM Targets, AircraftOperators
        WHERE TargetType = AircraftType
        GROUP BY Target, AircraftType HAVING COUNT (*) = 1.
```

Note that other queries can be constructed to achieve the same outcome. The result of this query executed for the example data is shown below, as Figure 11. In this case, the country of origin can be inferred for only three of the listed targets, because the other ship types and aircraft types of all the other targets are operated by more than one country.

Target	Target Type	Country
VN603	FFG07	AUS
VP337	MiG-29	MAL
VA920	F-16	USA

Figure 11: *Targets for which the possible country of origin is unique.*

It remains to more completely embody these ideas about the kinds of operations that the run-time engine should support. How these operations are expressed, together with a framework to specify when particular operations should be invoked, will form the basis of the intermediate code produced by the translator generator and interpreted by the run-time engine.

4. Specifying Behaviour

Notice that entire sequences of relational operators can be described by one SQL query. That is to say, what the translator generator regards as a whole sequence of relatively simple actions can be potentially expressed as a few SQL statements. Indeed, when an SQL query is submitted to a relational database management systems (RDBMS), it is decomposed into a relational calculus expression (or something similar). Further processing seeks to obtain a new expression for execution, equivalent to the original but incurring significantly lower computational cost.

The variation in the demand placed on system resources by different ways of expressing the query should not be under-estimated. Without even rudimentary query optimisation, databases of even moderate size simply could not operate in the real world. The problem of finding ways to execute transactions efficiently has attracted enormous and intensive research attention.

For this reason, it makes sense for the behaviour of the run-time engine to be specified using high-level constructs that can be easily expressed in SQL [16]. In this way, the engine would submit complete SQL statements to the underlying database system, and in doing so, take full advantage of the query optimisation that the transaction management services offer. In this way, the translator generator can be largely free to produce sequences of actions without too much regard for the computational costs they would otherwise incur. The only overhead will be the need to compose such action sequences into high-level statements after the translator generator proper has completed its search.

It is worth noting here that the query optimisation of the RDBMS is not a replacement for the ability of an on-line translator generator to gather global information, particularly in more sophisticated applications where the translator might have the ability to use the services of various available information systems. The RDBMS can only control the cost of individual queries, not the long-term operation of the translator as a whole.

In the emerging picture, a user supplies a schema and possibly other information to a translator generator, which produces as its output a specification of the behaviour of the translator. This is presented to the engine at run-time, which utilises the services of a database system to manipulate and store both temporary and predefined information. Incoming messages are parsed, and the resulting information presented to the engine, which places it in the database and initiates some processing which may ultimately produce some tables that can be composed into one or more output messages.

At this point, a reasonable description of the kinds of operations that the engine will perform has been established. It remains to connect these operations with the information in the database, by specifying a way of informing the engine about when the specific actions should occur.

Perhaps the starting point in resolving this issue would be to consider specifying the manipulations to be performed upon receiving each type of message. However, the actions that should occur will depend not only on the message received, but also on what has occurred in the past. This history is implicitly remembered as the information currently in temporary storage, so the action to occur must depend on the received message together with the current state of the database. Now remembering that the content of an incoming message is injected into the database upon arrival, the current state already embodies the new message at the moment when the engine will begin to apply its operations.

That is to say, the engine will decide what action it should take by observing the structure of the database immediately after the information from the new message has been absorbed. To see how this works, consider an incoming message stream consisting of two message types, one that specifies the position and time for a number of targets, and the other that specifies the target's type. Upon receiving the first of these two messages, the state of the database might appear as shown in Figure 12.

Target	Position	Time
VN603	2345S13245E	04-021234Z
VN159	2346S13250E	04-060213Z
VP717	2353S13421E	04-101351Z

Figure 12: *The state after receiving the first message.*

At this point, no action can be taken to generate an output message, because the output message stream consists of a single message type that requires longitude, latitude, time, and target type. At this point, actions might be applied to derive the latitude and longitude for each target, resulting in the replacement of the table of Figure 12 with the new table of Figure 13.

Target	Latitude	Longitude	Time
VN603	2345S	13245E	04-021234Z
VN159	2346S	13250E	04-060213Z
VP717	2353S	13421E	04-101351Z

Figure 13: *A new table derived to replace the first.*

No further manipulations can be made, and still no output message can be produced. However, some time later another message is received that contains information about target type, in this case only for two of the targets. The database now contains the new table of Figure 14 in addition to that of Figure 13.

Target	Target Type
VN603	Ship
VN717	Aircraft

Figure 14: *Some new information arrives.*

At this point, a natural join between the two tables produces a result that can be packaged as an output message. That is, an operation can occur that uses the information for targets numbered 'VN603' and 'VN717' to produce the table of Figure 15.

Target	Target Type	Latitude	Longitude	Time
VN603	Ship	2345S	13245E	04-021234Z
VP717	Aircraft	2353S	13421E	04-101351Z

Figure 15: The result of the natural join.

Notice that the details of the location of target 'VN159' were not used; that is to say, the result of the join does not include a row for this value of the join attribute. Therefore, the table of Figure 13 would be retained with just this single row, in the hope that a message specifying the type of the target 'VN159' will arrive in the future. Finally, all the rows of the table of Figure 15 would be removed (and therefore the table itself is deleted) as they are used in composing the output message.

Now consider what might happen if the two input messages were received in opposite order. Initially, information detailing each target's type is injected into the database, and the state at that moment consists solely of the table shown in Figure 14. No further processing can be performed until the second message arrives, when the state also includes the table in Figure 12. Now the operation to split position into latitude and longitude can occur, replacing the table of Figure 12 with that of Figure 13. Once this is

completed, the second operation can be performed to construct the result illustrated as Figure 15.

Observe that the output message and the final state are the same irrespective of the order in which the messages arrive. Furthermore, the operation to be performed at any moment depends only on the current state of the database. It is also emerging that each part of the specification for a translator will consist of more than just an SQL statement; an action involves retrieving data through the query, inserting this into another table, and removing related rows from some of the existing tables. Additional complications such as information that is available implicitly through stored procedures may need to be specifically supported in the engine.

Note that actions are not only performed when a new message arrives, but whenever an available action is applicable. In this sense, the arrival of a new message is much like performing some action: both merely result in a change in the state of the database. The only difference is that actions are under the control of the engine, whereas receiving a new message is an event that is imposed from outside. Generating an output message is simply an action that removes information from the database without replacing it with some derived fact. Indeed, the output message types will be declared to the engine in exactly the same way as any other data manipulation.

The behaviour specification is highly declarative: it indicates what the engine should do to translate incoming messages into the output language, rather than detailing how this is to be achieved. Even without the translator generator, this offers a high-level language for describing the translator in a reasonably natural way. The actions that the engine will execute are formalised versions of statements such as 'split the target's position into latitude and longitude', 'determine the latitude, longitude, time and type of each target', and 'find the targets that have only one possible country of origin'.

The engine will accept a set of rules, each of which contains a precondition in the form of a single query that describes the database state that must be attained for the rule to fire, and a list of actions to modify the database. The example above would contain a rule that generates the latitude, longitude, time, and type for some targets. The action for this rule would be to insert any such tuples into another table, while eliminating all rows from the source tables that contain a target that also appears in the result.

Consider the hypothetical rule shown below, which might be used to express the derivation used in the example earlier to produce an output message. The data modification statements to record the result and remove used information from the source tables are executed for each row generated by the query. If the query returns no rows, then no action is taken. Using a record of what tables are present in the database at any time, the engine will not even bother to test the query unless all of its source tables are known to exist and contain at least one tuple.

```
DEFINE ExampleRule AS
    SELECT Target, TargetType, Longitude, Latitude, Time FROM TargetLocations,
    TargetTypes
    WHERE TargetLocations.Target = TargetTypes.Target;
    INSERT INTO Targets VALUES $Target, $TargetType, $Longitude, $Latitude, $Time;
    DELETE FROM TargetLocations WHERE Target = $Target;
    DELETE FROM TargetTypes WHERE Target = $Target;
END
```

In this example, special identifiers have been used to explicitly refer to the attributes of the result of the query. In this way, the attribute 'Target' of the source table 'TargetLocations' is easily distinguished from the attribute 'Target' of the query result. It is worth noting that it is not actually necessary to do this, because the actions are automatically constrained to operate only within the context of the result of the query. That is, the restriction to delete only rows that also appear somewhere in the query result is implied, and the more concise form illustrated below might be considered instead.

```
DEFINE ExampleRule AS
    SELECT Target, TargetType, Longitude, Latitude, Time FROM TargetLocations,
    TargetTypes
    WHERE TargetLocations.Target = TargetTypes.Target;
    INSERT INTO Targets VALUES Target, TargetType, Longitude, Latitude, Time;
    DELETE FROM TargetLocations;
    DELETE FROM TargetTypes;
END
```

This more sophisticated approach arguably loses in clarity what it gains in conciseness. Most significantly, it places the demand on the engine's rule compiler to generate the appropriate restriction clause. In turn, this requires more effort to keep track of the primary keys of the source tables and the query result. Therefore, to simplify efforts at implementing the translator engine, explicit identifiers to refer to the attributes of the query result will initially be used.

The run-time engine should also manage the creation and removal of tables, thereby removing the need for such code to be explicitly written into the rules. Whenever an insertion operation is encountered, the engine should first check its index to ensure that the table being inserted into actually exists in the database. If not, then the engine is responsible for first creating the table. Likewise, after deleting from a table, the engine should also drop the table if it contains no tuples.

The example of Figure 10 presents a less trivial case, highlighting the need for the engine to support not only a simple query as the precondition of a rule, but any SQL statement that produces a single table. The union of two results produces a single table,

so a rule to generate a table of information about those targets for which the country of origin is unique might be as shown below.

```
DEFINE FindUniqueOwners AS
    SELECT Target, TargetType, Country FROM Targets, ShipOperators
    WHERE TargetType = ShipType
    GROUP BY Target, Country HAVING COUNT (*) = 1
    UNION
    SELECT Target, TargetType, Country FROM Targets, AircraftOperators
    WHERE TargetType = AircraftType
    GROUP BY Target, Country HAVING COUNT (*) = 1;
    INSERT INTO TargetCountry VALUES $Target, $TargetType, $Country;
    DELETE FROM Targets WHERE TargetType = $TargetType;
END
```

This example could be broken down into separate stages, by computing target data for ships and for aircraft separately, and then counting the number of rows relating to each target in another rule. This results in three simpler rules equivalent to the single rule above. Notice how deletion of the tuples in the 'Targets' relation is delayed until after both the first and second rules have been applied, ensuring the correct response when a ship type and an aircraft type have the same name.

```
DEFINE FindShipOwners AS
    SELECT Target, ShipType, Country FROM Targets, ShipOperators
    WHERE TargetType = ShipType;
    INSERT INTO TargetOwners VALUES $Target, $ShipType, $Country;
END

DEFINE FindAircraftOwners AS
    SELECT Target, AircraftType, Country FROM Targets, AircraftOperators
    WHERE TargetType = AircraftType;
    INSERT INTO TargetOwners VALUES $Target, $AircraftType, $Country;
END

DEFINE FormTargetResult AS
    SELECT * FROM TargetOwners
    GROUP BY Target, Country HAVING COUNT (*) = 1;
    INSERT INTO TargetResult VALUES $Target, $TargetType, $Country;
    DELETE FROM Targets WHERE Target = $Target;
END
```

The engine must also provide a mechanism for specifying that a table should be sent to the composer, where it will be used to formulate an outgoing message. The engine

would interpret some suitable non-SQL statement to mean that the resulting table should be sent to a symbolically named composer, where the symbolic name might correspond, say, to an IP address and port number. The idea behind the particular construction presented here is that this new statement should resemble an SQL insertion, in the interests of maintaining a consistent style throughout the rules.

```
DEFINE SendResultOff AS
    SELECT * FROM TargetResult;
    ACCEPT OutputComposer VALUES $Target, $TargetType, $Country;
    DELETE FROM TargetResult;
END
```

Other language constructs will surely be needed in the future, as the language evolves in the light of practical experience. In particular, some error handling and recovery mechanisms will be necessary, particularly with regard to constraint violations that faulty data might produce. This is an important subject for future work.

5. The Engine is a Production System

The engine instantiated with a behaviour specification is actually a production system, whose rules manipulate entire tables of data, rather than just single rows at a time.

A production system contains a buffer listing all conditions that are currently true, called the 'context'; in the case of the translator engine, the context is a list of all the tables currently in the database. An iteration of the algorithm starts by evaluating production rules against the context, with those rules whose preconditions are satisfied included in a 'hit list'. If the hit list is not empty, one rule is chosen for evaluation according to the conflict resolution strategy. Applying the rule will (normally) result in changes to the context, and the process continues until the hit list is empty.

The conflict resolution strategy, whereby a single rule is selected for execution from the hit list, is a fundamental issue. If the order in which the rules in the hit list (and any subsequent rules that might be enabled) are executed has no impact on the ultimate result, it does not matter how the engine selects from the hit list. The conflict resolution strategy is significant only when firing rules in a different order can change the overall behaviour of the system. Consider the two rules shown below.

```
DEFINE FindShipOwners AS
    SELECT Target, ShipType, Country FROM Targets, ShipOperators
    WHERE TargetType = ShipType;
    DELETE FROM Targets WHERE TargetType = $ShipType;
    INSERT INTO TargetOwners VALUES $Target, $ShipType, $Country;
END
DEFINE FindAircraftOwners AS
    SELECT Target, AircraftType, Country FROM Targets, AircraftOperators
```

```
        WHERE TargetType = AircraftType;
        DELETE FROM Targets WHERE TargetType = $AircraftType;
        INSERT INTO TargetOwners VALUES $Target, $TargetType, $Country;
    END
```

Suppose that these two rules are enabled at some moment, and that the table 'Targets' lists a target that could be either an aircraft or a ship. That is, suppose that some aircraft type and some ship type happen to share the same name, and that there is a target listed with this name as its type. The order in which these two rules are fired is then significant; if 'FindShipOwners' is fired first, say, then this target will be removed and therefore never processed by the 'FindAircraftOwners' rule.

The difficulty occurs because an intermediate result is required for more than one production. Two better solutions appeared in the previous section, in which either all the rules were combined into a single rule, or the delete statements were delayed until a third rule collates the results. This shows how such difficulties occur simply as the result of poor rule design.

Typical expert system shells [18,19], handle conflict in general by using some consistent technique for choosing a single rule from the hit list, and the rule designer is responsible for constructing the rule base with this particular choice in mind. Shortest rule first, priority or salience values, ordering based on specificity of the precondition, and random rule selection are all apparent in the expert system shells that are currently available.

If the order in which enabled rules fire is significant in terms of the result produced, this may represent poor rule design or inconsistency in the database. To address the first case, a better design usually amounts to representing the relationships between different rules explicitly in terms of additional facts in the database; indeed, an aim of the conceptual design process is to precisely qualify all such relationships before any rules are written. Proper representation and enforcement of database constraints, especially referential integrity, manages the second possible cause by detecting such inconsistencies as they occur.

Any use of priority or salience values to control rule execution is a practice to be discouraged in building any rule-based system, and the translator is no exception. As a result, no method for specifying priority weightings of any kind will be available. Circumstances do arise, however, when it is highly desirable to be able to specify that some rule can only be enabled after some set of other rules have first had a chance to fire. This kind of need often arises when detecting and processing exceptional conditions. For example, suppose that a ship can be identified either by its name or by its pennant number. Ships for which a name is provided might be processed by a rule such as

```
DEFINE ProcessNamedShips AS
    SELECT Target, Name, Class FROM ShipTargets, KnownVessels
    WHERE ShipTargets.Name = KnownVessels.Name;
    INSERT INTO VerifiedShipTargets VALUES $Target, $Name, $Class;
    DELETE FROM ShipTargets WHERE Target = $Target;
END
```

Another rule might process any naval targets that instead identify the vessel by its pennant number, deriving the name using some predefined information.

```
DEFINE ProcessNumberedShips AS
    SELECT Target, Name, Class FROM ShipTargets,
    KnownVessels
    WHERE ShipTargets.Pennant = KnownVessels.Pennant;
    INSERT INTO VerifiedShipTargets VALUES $Target,
    $Name, $Class;
    DELETE FROM ShipTargets WHERE Target = $Target;
END
```

In this case, it is actually not that difficult to construct a third rule to handle the case when neither the name nor the pennant number are provided or already exist in the 'Known Vessels' table. The action of the rule might be to record an error condition and delete the offending rows, or perhaps to update the table of known vessels.

```
DEFINE ShipsException AS
    SELECT Target FROM ShipTargets
        WHERE Name NOT IN (SELECT Name FROM KnownVessels)
            AND Pennant NOT IN (SELECT Pennant FROM KnownVessels);
    WRITE ErrorLogFile 'Cannot Identify Target ' $Target;
    DELETE FROM ShipTargets WHERE Target = $Target;
END
```

As the number of possible ways for the target to be identified increases, or as the queries become increasingly complicated, it is easy to see that the rule to detect and respond to the error would quickly become large and unwieldy. Even here, the query of the exception-handling rule contains two sub-queries. Permitting some way of specifying rule priorities leads to a more scalable and manageable solution.

Rather than introduce the ability to define priorities on rules, it is considered far better to allow a rule to explicitly state that it cannot be fired unless every member of a set of other rules is not currently enabled. This leads to an elegant and natural extension to the rule syntax, as illustrated in the improved exception-handling rule shown below.

```
DEFINE NewShipsException :- ProcessNamedShips, ProcessNumberedShips AS
    SELECT Target FROM ShipTargets
    WRITE ErrorLogFile 'Cannot Identify Target ' $Target;
```

```
        DELETE FROM ShipTargets WHERE Target = $Target;
    END
```

Note that the distinction between a rule being enabled and actually appearing in the hit list is important here; checking each rule against the current hit list will produce varying results depending on the order in which the rules are processed. That is to say, an enabled rule should only be inserted into the hit list if none of the named rules in its 'exclusion list' are also enabled.

A novel approach to conflict resolution that might be considered for the run-time engine is a conglomerated kind of rule execution: the queries for all rules in the hit list could be first evaluated, then all insertion statements could be performed, before any deletions are carried out. To realise a scheme such as this, some record of the tuples returned by each query precondition would be needed, perhaps in the form of a number of temporary tables that are removed at the end of the execution cycle. This is quite a radical departure from the traditional expert system shell. Because of this, it is not likely that the first implementation of the run-time engine will visibly operate this way, but the possibility should certainly be explored in the future.

A view of the overall architecture of the engine and its interfaces is provided as Figure 17. A behavioural specification provided by the user or the translator generator will be parsed to construct the rule base, and to initialise the context to reflect the predefined scenario dependent information in the database. At any moment, the context itself contains information about the current structure of the database, including foreign keys, subsets, exclusions, and other integrity necessary capabilities not directly supported in relational systems.

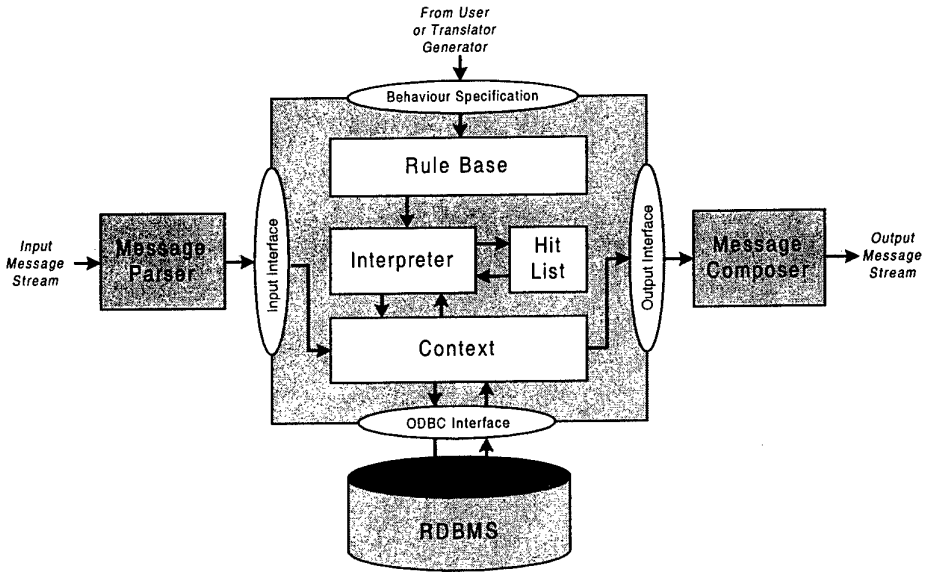


Figure 17: Overall architecture of the engine.

During execution, incoming messages are parsed and the results submitted to the engine through an input interface. The context is absorbs the new information, storing it in the underlying database. At the heart of the engine lies the interpreter, which examines the rule base with respect to the context to build a hit list of rules whose preconditions are met. When a rule from the hit list is fired, the context directs appropriate operations in the relational database to derive new relations, remove applicable temporary tuples from existing relations, and delete any empty tables that result.

At some moment, the context may satisfy the preconditions of a rule indicating that an output message can be formulated. When such a rule appears in the hit list, the interpreter directs the context to retrieve the specified information from the database and send it through the output interface to a message composer. As with any other rule, tuples in the result not marked as persistent will be subsequently removed from the database. The input and output interfaces must simply facilitate the transmission of relations named in the acceptance rule.

6. Some Other Useful Technologies

As described here, the run-time engine is built around a relational database system. This choice is based on development considerations alone; the same underlying principles could be equally applied with a different kind of data model in mind.

The relational model is not the only way to represent information in the engine. Some of the deficiencies of relational products are due to the fact that they do not properly support domains. If such support were implemented, an RDBMS could fully represent subset hierarchies. Another important limitation from the point of view of the translator engine is the presumption that all tables contain simple values. For example, the relationship between position and its corresponding latitude and longitude cannot be stored satisfactorily in a table of explicit values. Instead, a way of representing this implicitly as stored procedures is needed.

Object-relational database management systems (ORDMS) blend features of object-oriented database management systems (OODMS) and pure relational systems. In doing so, they provide some of the advantages of object-orientation, while still maintaining the strengths of relational databases. These emerging products provide features such as declarative referential integrity, complex data types and subset hierarchies, hierarchies of entire tables, and the ability to invoke methods and use their return values as columns. Such systems also allow values in a column to be cast from one type to another through predefined and user-defined casting operators.

Fully object-oriented databases are unlikely to be appropriate for implementing the engine, as they focus on complex interactions and large data types, and do not properly support the ad-hoc querying and data definition that the engine will require. It

currently appears that an object-relational system would provide capabilities best matched to the demands of the translator engine, but this would come at the possible price of reduced portability.

Appendix A. The Relational Model of Information.

A.1. Attributes, domains, tuples, and relations.

In the relational model of information, the database appears to the user or applications programmer to be composed of some number of multidimensional tables, or 'relations'. The format of a relation r is defined by its corresponding relation scheme R, which is a set of 'attributes' that name the columns of the table. The 'domain' $D(A)$ of any attribute A is the set of values that a corresponding column entry may assume.

Target Data		
Target	Target Type	Country
VN603	FFG07	AUS
VC202	F/A-18	USA
VP337	F/A-18	AUS
VZ990	FFG07	AUS
VA920	F-16	USA

Figure A1: *A relational table.*

Figure A1 illustrates a relation called 'Target Data', which has the relation scheme {'Target', 'Target Type', 'Country'}. Notice that the relation scheme is a set, so that the ordering of the columns is irrelevant. The names 'Target', 'Target Type, and 'Country' are attributes. The domain of 'Target Type', for instance, is a set that includes the values 'FFG07', 'F/A-18', and 'F-16', among others. Notice that so-called relational database systems do not fully support the relational model, because they do not properly represent domains.

An 'R-tuple' t is a function from the relation scheme R to the corresponding domains of the attributes forming R. This mapping can be restricted from the relation scheme R to any subset X of R, and the X-tuple so specified is usually denoted as $t[X]$. A relation r with relation scheme R is then defined to be a set of R-tuples.

In Figure A1, each row is a tuple on the relation scheme {'Target', 'Target Type', 'Country'}. The relation is a set of such tuples, so the order in which the tuples are displayed has no significance. Restricting the last tuple to relation scheme {'Target', 'Country'} produces the tuple in Figure A2.

VA920	USA

Figure A2: *The restriction of the last tuple.*

This definition can be restated in an alternative and equivalent form, by describing a relation r as a subset of the Cartesian product of the domains $D(A)$ associated with the attributes A forming the relation scheme R.

By definition, all the tuples t in a relation y must be distinct. Consider some subset X of the relation scheme R for which all X-tuples $t[X]$ are unique: $\forall t_i, t_j \in r \bullet t_i[X] \neq t_j[X]$. This is trivially true for $X=R$, so there must always be at least one such subset for every possibly relation. In addition to this, the relation schemes of most relations will be one or more proper subsets $X \subset R$ that satisfy this property. Any such a set X for which all X-tuples are distinct is called a 'superkey' of the relation. If the superkey X also has the property that removing any attribute from X leaves a set of attributes Z that is not a superkey, then X is said to be a 'key' of relation r.

For example, the relation of Figure A1 has no rows with the same value for both the 'Target' and 'Target Type' attributes, so {'Target', 'Target Type'} is a superkey of the 'Target Data'. However, removing 'Target Type' produces the set {'Target'}, and no two values for this attribute are repeated. Therefore, {'Target', 'Target Type'} is not a key. In fact, the set containing the single attribute 'Target' is the only key of relation 'Target Data'. This means that the value of 'Target' can be used to uniquely identify any tuple in the relation.

In general, there may be more than one key to a relation, and usually one of them will be designated as the 'primary key'. It is conventional to underline the set of attributes forming the primary key, as was the case in Figure A1.

A relational database scheme is a set of relation schemes $S = \{R_i \mid i = 1,\ldots,n\}$ together with a set of integrity constraints C. An instance of this database scheme is a database, defined to be a set of relations $s = \{r_i \mid i = 1,\ldots,n\}$ for which the relation scheme of each relation $r_i \in s$ is $R_i \in S$ and that together satisfy the integrity constraints C.

A.2. Some Integrity Constraints.

Various kinds of integrity constraints can be defined on the database scheme. The 'entity integrity constraint' specifies that no primary key can contain the null value, because this would violate the property that every tuple can be identified by its primary key value. 'Key integrity' constraints demand that all candidate keys be unique for every tuple of the relation. Referential integrity constraints indicate that tuples of one relation should refer to an existing tuples in another, and are specified in terms of foreign key relationships.

Consider two relations $r_i \in s$ and $r_j \in s$ having relation schemes $R_i \in S$ and $R_j \in S$, respectively, and suppose that the primary key of relation r_j is the set of attributes $Y \subseteq R_j$. A set of attributes $X \subseteq R_i$ is said to be a 'foreign key' of r_i if two conditions are met. Firstly, X must have the same set of domains as the primary key Y of r_j. Secondly, for each non-null tuple $t_i \in r_i$, there must exist some tuple $t_j \in r_j$ with $t_i[X] = t_j[Y]$.

For example, consider a relation 'Vessels' that stores information about the name, pennant number, and class of each ship. Because in this case each ship can be identified by either name or pennant number, there are two candidate keys, and either of these

could be selected as the primary key. A second table 'Ship Data' lists the type of each ship class.

It is sensible to demand that that all ship classes have a ship type recorded for them. To enforce this, any ship class specified in 'Vessels' should be first recorded as a ship class in the 'Ship Data' relation. That is, the 'Ship Class' attribute of 'Vessels' is a foreign key referring to the 'Ship Data' relation. According to convention, this referential integrity constraint is indicated in the database scheme of Figure A3 by the directed arc connecting the foreign key 'Ship Class' of 'Vessels' to the key 'Ship Class' of the referred relation 'Ship Data'.

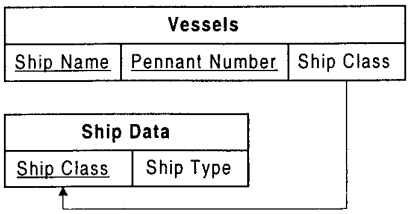


Figure A3: A small database scheme with a referential integrity constraint.

Notice also that a foreign key can refer back to its own relation; this can occur when a relation contains two attributes that have the same domain. While most relational database systems enforce entity and key integrity, few support foreign key relationships, and so it is usually left up to the applications layer to ensure referential integrity.

Databases will typically require other kinds of constraints to preserve their integrity. Attribute constraints include such things as specification of whether null values are allowed in the column, and this is available in most relational database systems. More general relationship structural constraints, of which referential integrity constraints are a special case, may be required. Subset and superset constraints qualify the relationships between various specialisations and generalisations of entities.

Such semantic integrity constraints are supported to varying degrees by different relational database systems. For instance, some systems provide the ability to declare static constraints as assertions. For example, a constaint to specify that ship targets might appear something like the statement below.

```
ASSERT ShipConstraint ON Targets, ShipTargets
  AS Targets.Target = ShipTargets.Target WHERE Targets.TargetType = 'Ship';
```

Note that this is not necessarily syntactically correct for any particular relational system; it is merely intended to illustrate the general idea. Some systems also provide trigger mechanisms to perform some specified procedure when a condition is met, and this can be used to maintain semantic integrity.

```
DEFINE TRIGGER ShipTrigger ON ShipTargets, Targets
  AS ShipTargets.Target NOT IN Targets.Target
```

```
INSERT INTO Targets(Target, TargetType) VALUES (ShipTargets.Target, 'Ship');
```

Again, the example is not indicative of the syntax of any particular system that supports such a mechanism.

A.3. The Relational Algebra.

The relational algebra is a formal definition of operations that manipulate entire relations. Each relational operator performs an action on some number of argument relations, and produces a new relation as its result. A relation is defined to be a set of tuples, so the first operations that can be included are set operations such as union, intersection, set difference, and Cartesian product. Other operations such as selection, projection, and join were specifically developed within the theory of relational databases.

The select operator is used to single out the subset of the tuples in a relation that satisfy a given boolean condition, and is denoted as $\sigma_{Condition}(relation)$. For example, to select the targets from the 'Target Data' table of Figure A1 that belong to Australia, the operation $\sigma_{Country='AUS'}(\text{TargetData})$ would be used. The relation produced by this operation is illustrated in Figure A4.

Target	Target Type	Country
VN603	FFG07	AUS
VP337	F/A-18	AUS
VZ990	FFG07	AUS

Figure A4: The result of the selection operator.

The projection operator, denoted as $\pi_{Attributes}(relation)$, is defined when the set of attributes is a subset of the relation scheme of the specified relation. The result is the set of distinct tuples produced by restricting the tuples of the argument relation to the named attributes. As an example, the projection operator $\pi_{\text{TargetType, Country}}(\text{TargetData})$ produces the relation of Figure A5.

Target Type	Country
FFG07	AUS
F/A-18	USA
F/A-18	AUS
F-16	USA

Figure A5: The result of a projection operator.

The join operator is used to connect information disseminated across several tables. The join operator is denoted[1] as $relation_1 \otimes_{Condition} relation_2$, and produces as its result the set of tuples from the Cartesian product of the two argument relations that satisfy the given join condition. For example, consider the first relation 'Aircraft Data' in Figure

[1] Note that the ⊗ symbol used here is not the standard notation; the more usual bowtie symbol was not available in preparing this document.

A6, and again the relation of Figure A1. The result of the join operator $\text{TargetData} \otimes_{\text{TargetType=Aircraft Type}} \text{AircraftData}$ is illustrated as the second relation of Figure A6, overleaf.

Aircraft Data	
Aircraft Type	Category
F/A-18	F
F-16	F
F/A-18	A

Target	Target Type	Country	Aircraft Type	Category
VC202	F/A-18	USA	F/A-18	F
VC202	F/A-18	USA	F/A-18	A
VP337	F/A-18	AUS	F/A-18	F
VP337	F/A-18	AUS	F/A-18	A
VA920	F-16	USA	F-16	F

Figure A6: *Another relation and the result of a join.*

This is an example of an equijoin, in which the join condition is a simple equality requirement. Notice that two identical columns appear in the result; one of these can be eliminated using a projection operator. In this case, the combined join and projection would be expressed in the relational algebra as $\pi_{\text{Target,TargetType,Country,Category}}(\text{TargetData} \otimes_{\text{TargetType=AircraftType}} \text{AircraftData})$, which produces instead a relation without the redundant 'Aircraft Type' column.

This composition of equijoin and projection extremely common, and as a result, is specially defined as the natural join operator, denoted as $\text{relation}_1 \otimes \text{relation}_2$. The implied join condition is equality between foreign keys of each relation that refer to the other relation.

Appendix B. An Overview of Object-Role Modelling.

The ORM diagram is a graphical language typically used to express the conceptual structure of a database system, and it is within this context that it will be described here. Each real-world concept or object is termed an 'entity', and a set of related entities is an 'entity type', which is indicated on the diagram by an ellipse containing the name of the entity type. Entities are interrelated through 'fact types', each of which is comprised of some number of conforming 'facts'. Individual entities and facts are not explicitly represented on the diagram; instead they are organised into sets of related entities and structurally similar facts. For example, it might be asserted that pilots might fly aircraft.

There is a distinction between a real-world object and the way in which it might be encoded as a data value within the system. This is embodied in ORM by using a 'label type', which is indicated by a broken ellipse, to define the set of data values that will be used to uniquely identify each entity in a related entity type. Pilots might be identified by their names, and aircraft identified by aircraft numbers, for instance.

Any fact type is composed of a number of roles, which are drawn as boxes on the diagram and connected to the relevant entity type to indicate that at least some of the entities forming that entity type can play this role. The 'arity' of a fact type is the number of roles that it contains: on an ORM diagram, a sequence of n contiguous role boxes describes an n-ary fact type. Figure B1 illustrates a simple relationship between two entity types 'Pilot' and 'Aircraft', which are to be identified using 'Pilot Name' and 'Aircraft Number', respectively; all the fact types here are binary.

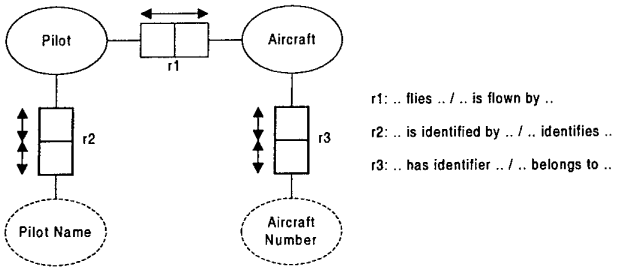


Figure B1: *Pilots and Aircraft.*

Arrows are used to indicate the existence of a uniqueness constraint on a given role or set of roles. This means that any combination of entities can appear in the set of roles spanned by the uniqueness constraint at most once. The uniqueness constraints on fact type 'r2' together describe a *1:1* relationship between pilots and their names. On the other hand, the single uniqueness constraint spanning both role boxes of 'r1' is the weakest possible constraint, allowing any combination of pilot and aircraft in an *m:n*

mapping. Because certain uniqueness constraints clearly imply other, weaker, constraints, only the strongest constraints are usually shown on the diagram.

Figure B2 represents a more convenient form of this first diagram. Here, the label types used to identify corresponding to each entity type are indicated in parentheses underneath the name of that entity type. In many cases, the label type is omitted altogether.

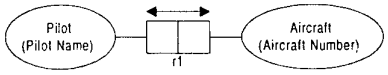


Figure B2: *An abbreviated and equivalent representation.*

Suppose now that pilots are assigned to a specific airbase. This introduces a new entity type 'Airbase' which might be identified by an airbase name (note that airbase names and pilot names are distinct label types). Each airbase might have many pilots assigned to it, so this represents a *1:m* relationship. Figure B3 shows the new schema diagram.

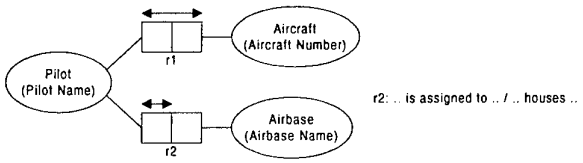


Figure B3: *Some more constraints.*

A role is said to be 'mandatory' (or 'total') if every entity in the connected entity type must participate. This is indicated on the diagram by placing a heavy dot where the connector meets the entity type, as in Figure B4. Here, each pilot must be stationed at an airbase (together with the uniqueness constraint on 'r2', this means that every pilot is stationed at exactly one airbase).

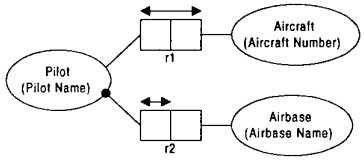


Figure B4: *Every pilot must be stationed at a unique airbase.*

It is also possible to connect more than one role connector to the same mandatory role dot to form an explicit role disjunction; this indicates that at least one of the applicable roles must be played by every entity in the entity type. Figure B5 instead demands that every pilot must either fly some aircraft or be assigned to some particular airbase, or both.

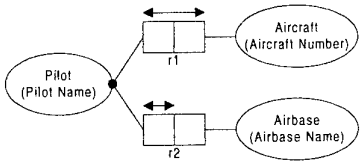


Figure B5: *An explicit role disjunction.*

Ternary fact types are also possible. Figure B6 is results from the requirement that the number of hours each pilot has spent flying any aircraft is to be recorded. That is, the combination of pilot and aircraft functionally determines a time in hours.

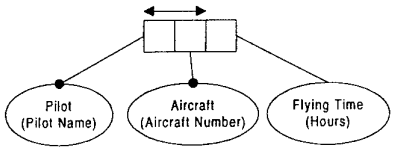


Figure B6: *A ternary relationship.*

Note that the uniqueness constraint spans two of the three roles that form this fact type. In general, an *n*-ary fact type should have no uniqueness constraints that span fewer than *n-1* roles; otherwise, the fact type can be split into several fact types of lower order.

It is often convenient to represent this kind of structure by using a 'nested fact type' to objectify the relationship between pilot and aircraft. In doing so, each combination of pilot and aircraft is treated as an entity in its own right. Note that the nested fact type itself must always have a uniqueness constraint spanning all of its roles; otherwise the entire structure could be decomposed. The ORM diagram of Figure B7 is equivalent to that of Figure B6.

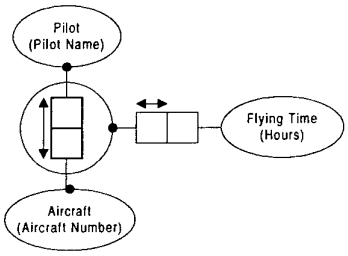


Figure B7: *A nested fact type.*

Another feature of ORM is its ability to represent subtype/supertype relationships. An arrow connects the subtype to its generalisation, and one or more fact types are connected to the supertype to define the specialisation. Figure B8 illustrates two subtypes 'Surface Vessel' and 'Submarine' of an entity type 'Vessel', with the membership of each subtype defined using the fact type that assigns to each vessel a 'Type'. The label type 'Type Code' consists of the two values 'surface' and submarine.

Each subtype plays different roles; otherwise there would be no reason for introducing the subtype in the first place. Any roles that all subtypes may play would be attached to the supertype, while each subtype is connected to roles that are specific to that subtype. Here, the primary radar is recorded for surface vessels, while submarines possess a primary sonar system instead.

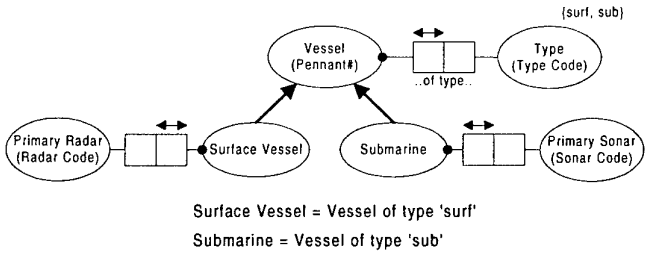


Figure B8: *A simple subtype structure.*

Note that subtypes may be further specialised; for instance, surface vessels might be divided into destroyers and frigates if different roles are played by each. While this example illustrates a simple partitioning of vessels into either one subtype or the other, it is also possible to construct type hierarchies in which a subtype might have multiple supertypes.

Numerous other kinds of constraints can also be expressed on the ORM diagram, including equality, subset, uniqueness, and exclusion constraints that operate between fact types, occurrence frequency constraints and entity type constraints. Most of these are beyond the scope of this overview; the reader is instead referred to [16] for a more complete discussion.

Appendix C. From ORM to a Relational Schema

The ORM diagram is a conceptual description of the structure of a universe of discourse. Once a design is finalised, the high-level description must be somehow implemented using the kinds of database systems that are currently available. In particular, relational systems of one form or another still dominate the database market, so it is natural to consider implementation as a relational database scheme.

The method by which the fact types of an ORM schema are grouped to form relational tables is called the Optimal Normal Form (ONF) algorithm. The result is a minimal set of relations that has no repeating attributes, no redundancy or update anomalies, and no dangling tuples. That is, the ONF algorithm yields a relation scheme in fifth normal form and having fewest relations.

The ONF algorithm is very simple. Each fact type with a composite key is implemented as a single table. All fact types with simple keys attached to a common object type are grouped into a single table, with the attribute corresponding to this object type as the key.

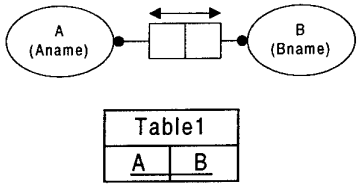


Figure C1: *A fact type with a composite key.*

Figure C1 shows a fact type having a composite key, so a single relation with attributes corresponding to the two object types is generated. Figure C2 illustrates several fact types having as their keys a single common object type, they are all grouped into the one table. Because the role box connected to object type 'A' of the lower right fact type is not mandatory, the corresponding 'E' column of the table stipulates that null values are permitted.

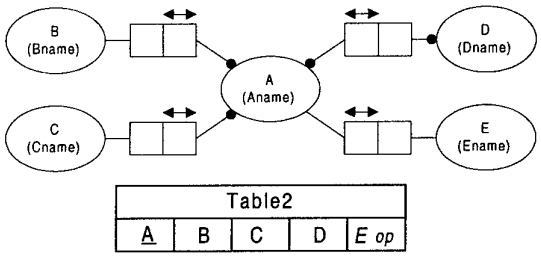


Figure C2: *Grouping simple keys into a relation.*

Binary fact types with two simple keys represent a one-one relationship between the involved entities. The usual way of handling this is to group the fact type into a table with other fact types connected to the entity type for which the corresponding role is mandatory. Figure C3 illustrates this.

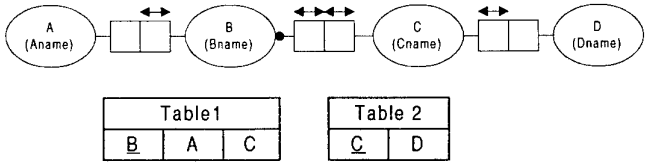


Figure C3: *Grouping with the mandatory role.*

If instead the role of entity type 'C' were mandatory, then the fact type having two simple keys could be grouped either way. A nested fact type, in which each fact is itself considered to be an object, is illustrated in Figure C4. The nested fact type is treated the same as any other fact type, except that it is represented in its relations by the combination of its defining object types.

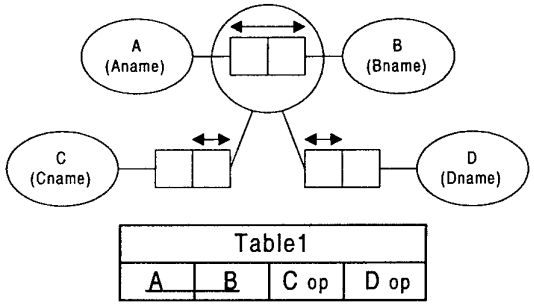


Figure C4: *Handling nested fact types.*

Relational systems provide no direct support for subtype hierarchies. Essentially, there are two ways to handle the subtype constraints of an ORM diagram: either the hierarchy is collapsed back into the base object type, or separate tables are created for each subtype. Figure C5 shows a simple subtype hierarchy, the result of collapsing these subtypes back into the base, and the table this produces.

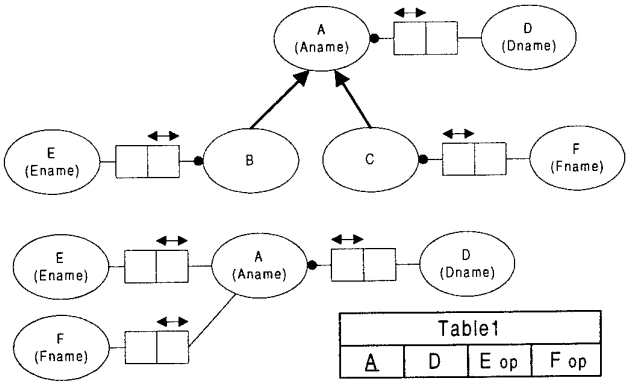


Figure C5: *Collapsing a subtype hierarchy and the table it produces.*

Additional assertions or triggers are required if the subtype relationship is to be properly enforced in the relational database. If instead separate tables were created for each subtype, the result would be the three tables of figure C6. Notice that the referential integrity constraints together capture the subset relationship, but some form of triggers or assertions would be required for the relationship that defines the subtype.

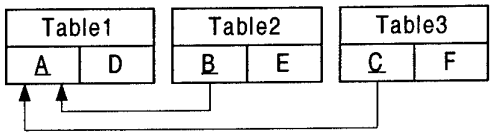


Figure C6: *Separate tables for each of the subtypes.*

Each approach has its advantages and disadvantages; the first option reduces the number of tables that are generated, but may produce much larger tables that are full of null values. In practice, it is common for a combination to be used, even within the same subset hierarchy.

Appendix D. An Example Translator Specification.

The example of this section illustrates a translator for two hypothetical message formats. While none of the message types described are directly taken from any real message formats, they accurately depict features typically seen in both the OTH-T Gold [26] and ADFORMS standards.

D1. An Input Message Format.

Every message type consists of a number of smaller units called 'sets'; in turn, each set is composed of some number of fields that contain simple data values. All message types share the same first and last sets, named 'MSGID' and 'ENDAT', respectively. The 'MSGID' set contains a field naming the message type, another giving the name of the sender, and one each for the position of the sender and the date and time at which the message was sent. All of these fields are mandatory.

MSGID / Message Type / Sender Identification / Position / Date-Time //

The 'ENDAT' message indicates that the end of the message has been reached. It also contains a number identifying the message, although it is not needed in this example.

ENDAT / Message Number //

The air report 'AIRREP' message type consists of a 'MSGID' set with the message type field containing the 'AIRREP' keyword, followed by a repeating group containing two the 'AIRCRAFT' and 'POSITION' sets, and finally ending with the 'ENDAT' set. The repeating group allows details about several aircraft to be provided within the same message.

The 'AIRCRAFT' set specifies a mandatory target number assigned to the aircraft, an optional aircraft type an optional aircraft category, and a mandatory country of origin. A special value indicates that the country owning the aircraft is not known.

AIRCRAFT / Target Number / Aircraft Type / Aircraft Category / Country //

The 'LOCATION' set conveys the bearing and direction of the target specified in the preceding set from the sender of the message, at the time when the message was sent.

LOCATION / Bearing / Direction //

The 'SEAREP' message contains information about ship targets, both surface and submarine. It consists of a 'MSGID' set followed by a repeating group containing a 'VESSEL' set and a 'LOCATION' set, with an 'ENDAT' set to indicate that the message is finished. The 'VESSEL' set defines the target number, giving the pennant number and name of the vessel, the ship's class, and its country of origin. The target number and country are mandatory, and the ship type must be given in the absence ship name, pennant number, and ship class.

VESSEL / Target Number / Pennant Number / Ship Name / Ship Class / Country //

The 'HOSTCONT' message type describes engagements with targets. It consists of the 'MSGID' set where the message identifier field contains 'HOSTCONT', followed by a repeating group containing the 'ENGAGE' set and the 'LOCATION' set, and finally the 'ENDAT' set. Note that this message type is highly abbreviated; in a real message format a great deal more information about the engagement would be represented. Here, the 'ENGAGE' set contains a mandatory target number and a value to indicate the type and outcome of the engagement.

ENGAGE / Target Number / Engagement Type / Engagement Result //

Ideally, any target listed in a 'HOSTCONT' message will have appeared previously in some 'AIRREP' or 'SEAREP' message that provides details about the aircraft or ship. Nonetheless, the translator will be able to cope if messages are presented to it out of order.

In summary, the input message format consists of the three message types 'AIRREP', 'SEAREP', and 'HOSTCONT'. The 'AIRREP' message type assumes the form

MSGID / AIRREP / Sender Identification / Position / Date-Time //
AIRCRAFT / Target Number / Aircraft Type / Aircraft Category / Country //
LOCATION / Bearing / Direction //
ENDAT / Message Number //

where the two middle sets form a repeating group. The complete 'SEAREP' message type

MSGID / SEAREP / Sender Identification / Position / Date-Time //
VESSEL / Target Number / Pennant Number / Ship Name / Ship Class / Country //
LOCATION / Bearing / Direction //
ENDAT / Message Number //

uses a 'VESSEL' set in place of the 'AIRCRAFT' set to instead convey information about a ship. Finally, the 'HOSTCONT' message type

MSGID / HOSTCONT / Sender Identification / Position / Date-Time //
ENGAGE / Target Number / Engagement Type / Engagement Result //
LOCATION / Bearing / Direction //
ENDAT / Message Number //

contains a list of information about engagements with various targets.

The basic information about targets that this message format represents is shown in Figure D1. A 'target' is conceptually an aircraft or ship as perceived by the unit that observes and reports it, so that different target numbers may be allocated to the same vessel or aircraft by different observing units. The combination of the sender and the target number uniquely identifies the observation. If only one unit is to be connected through the translator, then target number alone would be sufficient.

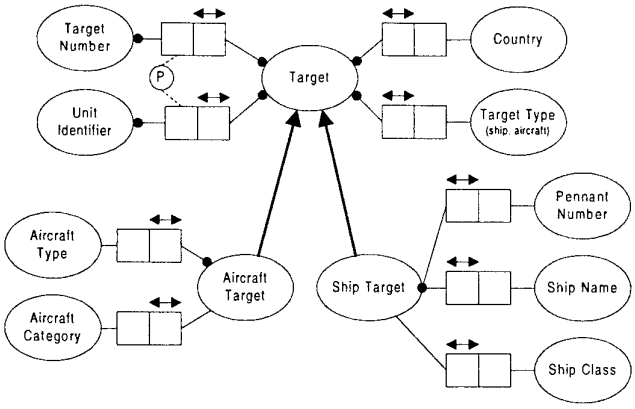


Figure D1: *Part of the structure of the input message format.*

Note that the mandatory role constraints on fact types involving aircraft and ship targets will enforce the requirement that this data be provided before the target can be described in a 'HOSTCONT' message. To permit the 'HOSTCONT' message to be received before 'AIRREP' or 'SEAREP' messages provide details about the target, these constraints should be relaxed.

Whenever a target is reported, the bearing and distance from the reporting unit is recorded. The position of the observer is contained in the 'MSGID' set that precedes the list of targets. Assuming that no object can be in more than one location at any moment, the combination of sender and date-time uniquely determines its position. Likewise, the combination of target and date-time should uniquely determine its bearing and distance, remembering here that target is a combination of a target number and the name of the reporting unit.

Proceeding along these lines would yield a structure that has two nested entities, one for the combination of unit identifier and date-time, and the other representing the combination of target and date-time. In addition, every date-time value recorded for a particular target must be one of the date-time values that are recorded for the specific unit identifier that reported the target. This qualified subset constraint, marked with '*' in Figure D2, is difficult to capture.

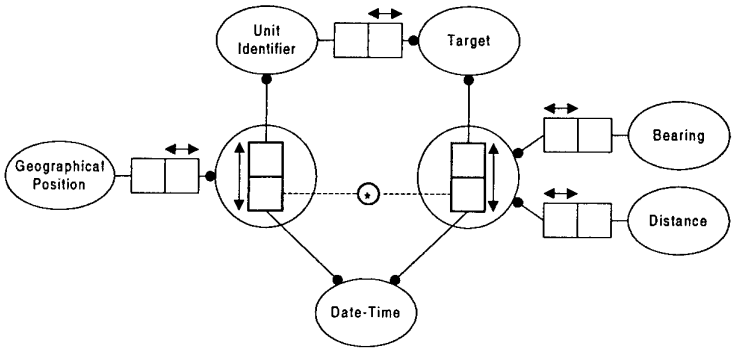


Figure D2: *One way of representing location information.*

A simpler model can be formulated by considering the position of the observer unit to be an attribute of the target rather than the observer itself. This is like repeating the value provided in the header for each target listed in the message, effectively collapsing the left-hand side of Figure D2 into the right. This corresponds to an augmentation of the functional dependency between unit identifier, date-time and geographical position with the additional target number attribute. The resulting structure is shown in Figure D3, and this diagram also includes the engagement information.

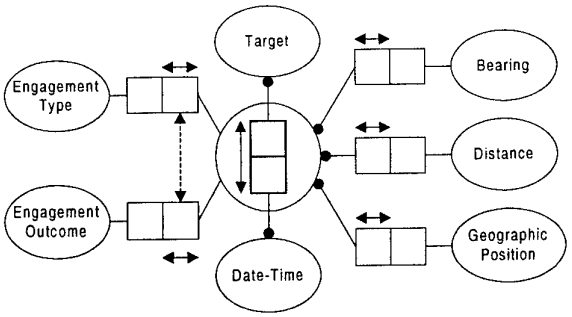


Figure D3: Engagement and location information.

In database terms, the cost of doing this is an increase in the table size, because the position value will be repeated for every target in a message, instead of being recorded once for the observer unit. However, there is no longer any need to represent the difficult subset constraint.

The incoming message stream is described by the combined structures of Figures D2 and D3. The corresponding relational schema illustrated in Figure D4 represents the various tables that will be injected into the database, as messages are received. Additional membership constraints resulting from the subtype relationship between 'Targets', 'Aircraft Targets', and 'Ship Targets' are not shown. Another constraint is also needed to enforce that a non-null value is provided for at least one of the columns 'Pennant Number' and 'Ship Name' of 'Naval Targets'.

Normally, all of the bearings, distances, observer positions and engagement data shown in figure A15 would be grouped into the same table, with the engagement type and outcome columns optional. The equality constraint involving engagement type and outcome is more easily captured by splitting the structure across the two tables 'Target Locations' and 'Engagements', with a subset constraint from the key of the second to that of the first. The engagement type and outcome columns are then mandatory in 'Engagements', thereby enforcing the equality requirement.

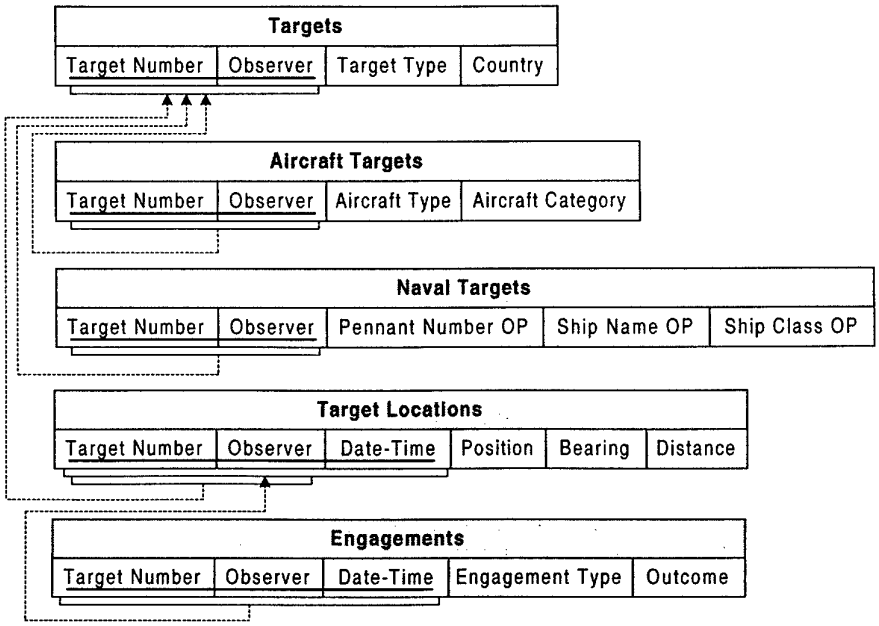


Figure D4: Information received from the input message stream.

Reception of an 'AIRREP' message will result in new information in the form of the 'Targets', 'Target Locations' and the 'Aircraft Targets' tables. Similarly, a 'SEAREP' message results in additions to tables 'Targets', 'Target Locations' and 'Naval Targets'. A 'HOSTCONT' message will generate appropriate information in the 'Targets', 'Target Locations' and 'Engagements' tables. If any integrity constraints are violated by the incoming information, the message should be rejected with an error message recorded in a log file, or perhaps displayed.

D2. Some predefined and scenario data.

In addition to the information received from the incoming message stream, some predefined facts and useful scenario-dependent data might be available to the translator. In essence, this will provide the glue that binds the information structure of the input format to that of the output format. For instance, suppose that functions are available to relate each date-time value to corresponding combinations of date and time. The ORM diagram that expresses this is shown as Figure D5.

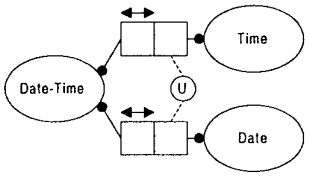


Figure D5: Relating date-time to separate data and time values.

The position of the observer and the bearing and distance of a target relative to this observer can also be used to compute the explicit position of the target. The relationships between observer position, bearing, distance and target position are shown in Figure D6. Also illustrated is the definition of position as a combination of latitude and longitude.

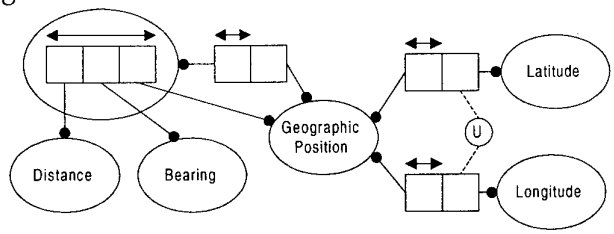


Figure D6: *Distances, bearings, and positions of a target and its observer.*

Suppose also that the aircraft category is known for each aircraft type, and that the combination of aircraft category and country of origin is sufficient to determine a suspicion code denoting the potential threat that the aircraft presents. This structure is illustrated in Figure D7.

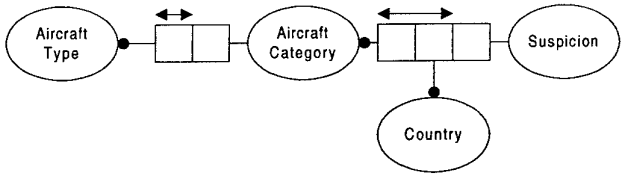


Figure D7: *Predefined aircraft data.*

Likewise consider predefined information about various ships that might appear in the scenario. Suppose, for example, that pennant numbers are not necessarily unique, but that the combination of pennant number and the owning country is sufficient to identify any particular vessel. Suppose also that names of ships are globally unique. Each ship belongs to a ship class, and every ship class belongs to a single category. Furthermore, the category of a ship and its country of origin together determine a suspicion code. This results in the ORM diagram of Figure D8.

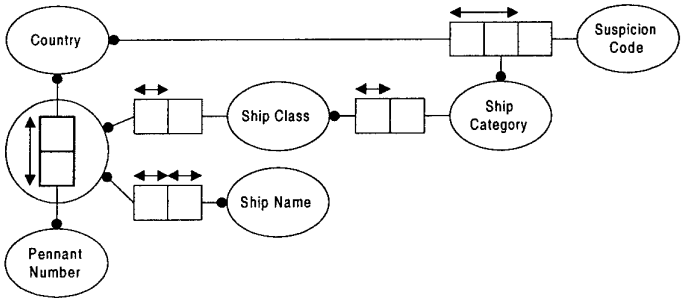


Figure D8: *Predefined information about ships.*

Suppose also that it is possible to assign to each target a unique identifier, say by defining a procedure to allocate consecutive numbers to each combination of target number and observer. This leads to a structure as shown in Figure D9.

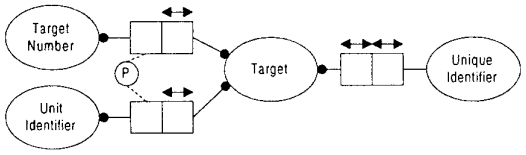


Figure D9: *A unique identifier for each reported target.*

The way in which engagement information is represented can vary widely between different message formats, even between different message types that belong to the same format. Suppose engagement type and outcome values can be combined with a suspicion code to yield a coded description of the engagement, which might ultimately be used by a composer to generate a simple sentence, say. Figure D10 illustrates this relationship.

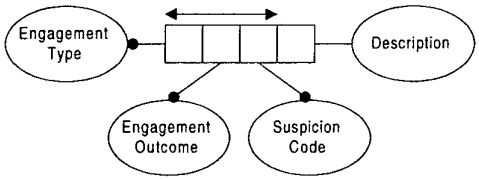


Figure D10: *Relating engagement information.*

The relational schema to store all of this predefined data appears in figure C11. Note that this is a high-level view only, and does not specify how these tables will actually be stored in the database. That is, there are an infinite number of rows in the 'Relative Position' and 'Dates and Times' relations, so they cannot be stored in the database as explicit tables for any realistic application. New object-relational systems offer the ability to represent stored procedures as columns in tables, which would make such complications invisible to the translator engine. However, older purely relational databases do not provide this service, so perhaps the engine itself should ultimately recognise that some columns may actually represent a function that needs to be invoked to access a required value. However, implementing this is not a simple task.

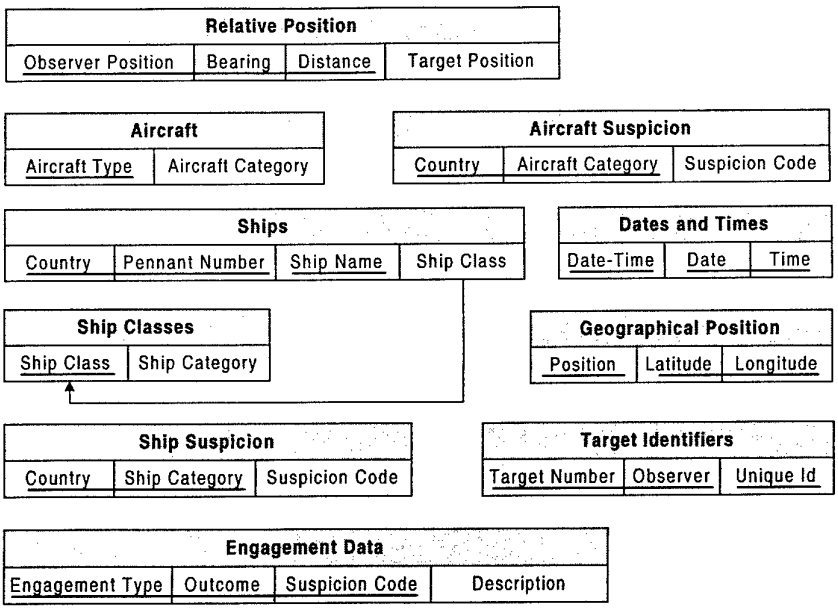


Figure D11: The relational schema for storing the predefined data.

The relational tables of Figure D11 will form the component of the database that is fixed in structure. Note that some applications will demand that the data in some of these tables can be altered, but the relation schemes will normally remain unchanged.

D3. An Output Message Format.

The output message format will consist of a single message type, which is more complex in structure than those of the input message format described earlier. In fact, the output message format is modelled on the very complicated OTH-T Gold CONTACT REPORT message.

The output message type contains a repeating group within another repeating group. The outermost of these forms a list of targets, each of which must be identified by a globally unique value. For each of these targets, the inner repeating group forms a list of positions and optional engagement descriptions.

```
MSGID / CONTACT / Sender Identification //
    TARGET / Unique Identifier / Name / Type / Category / Country / Suspicion Code //
        POSITION / Latitude / Longitude / Date / Time //
        ENGAGEMENT / Description //
ENDAT //
```

The 'TARGET' set contains information about both naval and air targets. For a ship, the 'Name' field is the name of the ship, 'Type' contains the ship's class, and 'Category' contains the ship's category. For aircraft, the 'Name' field is empty, 'Type' contains the aircraft type, and 'Category' is the aircraft category. In either case, the country of origin and the suspicion code are mandatory.

Each target has an associated list of positions (at least one must be provided), and values must be provided for data, time, latitude and longitude. An optional description of any engagement that occurred at this position may also be provided.

The conceptual schemata are not shown; Figure D12 illustrates a relational schema describing the output message format. An alternative exists that has only two tables, where an optional 'Description' column is added to the 'Output Locations' table to allow the 'Engagement Descriptions' table to be removed.

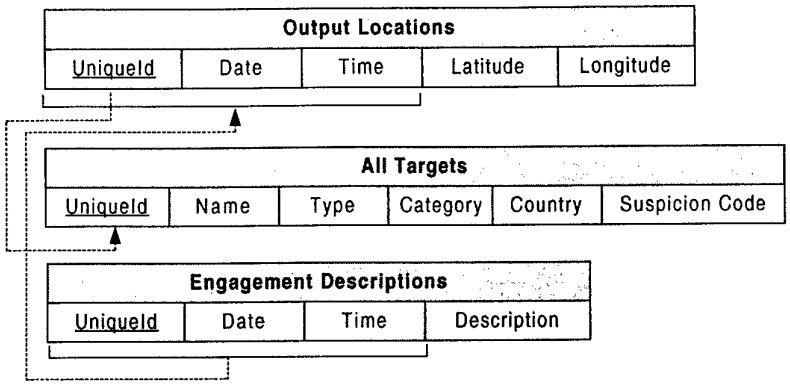


Figure D12: The relational schema of the output message format.

D4. The rules for translation.

The location of every target is provided in the incoming message stream as a bearing and distance relative to the position of the observer. The output format requires that target positions be given as a combination of latitude and longitude. The following rule operates on the 'Target Locations' table provided by the incoming message stream, using the predefined data tables 'Relative Position' and 'Geographical Position' to produce a new table that contains the latitude and longitude of each target.

```
DEFINE FindAbsolutePosition AS
    SELECT TargetNumber, Observer, Date-Time, Latitude, Longitude
    FROM TargetsLocations, RelativePosition, GeograpicalPosition
    WHERE TargetLocations.Position = RelativePosition.ObserverPosition
        AND TargetLocations.Bearing = RelativePosition.Bearing
        AND TargetLocations.Distance = RelativePosition.Distance
        AND RelativePosition.TargetPosition = GeograpicalPosition.Position;
    INSERT INTO TargetLocations1
    VALUES $TargetNumber, $Observer, $Date-Time, $Latitute, $Longitude;
    DELETE FROM TargetLocations
    WHERE TargetNumber = $TargetNumber AND Observer = $Observer;
END
```

The combined date-time field of the input message format also needs to be converted into separate date and time values. At the same time, the following rule replaces the target number and observer by a unique identifier.

```
DEFINE ModifyLocations AS
    SELECT TargetLocations1.TargetNumber, TargetLocations1.Observer,
        UniqueId, Date, Time, Latitude, Longitude
    FROM TargetsLocations1, DatesAndTimes, TargetIdentifiers
    WHERE TargetLocations1.Date-Time = DatesAndTimes.Date-Time
        AND TargetLocations1.TargetNumber = TargetIdentifiers.TargetNumber
        AND TargetLocations1.Observer = TargetIdentifiers.Observer;
    INSERT INTO OutputLocations
    VALUES $UniqueId, $Date, $Time, $Latitute, $Longitude;
    DELETE FROM TargetLocations1
    WHERE TargetNumber = $TargetNumber AND Observer = $Observer;
END
```

The next few rules are all concerned with processing information about aircraft targets. The first replaces the target number and observer combination by the unique identifier, to constructs a new table that lists for each such target its type, category, and country of origin.

```
DEFINE ProcessAirTargets AS
  SELECT TargetNumber, Observer, UniqueId, AircraftType, AircraftCategory, Country
      FROM Targets, AircraftTargets, TargetIdentifiers
      WHERE Targets.TargetType = 'Aicraft'
          AND Targets.TargetNumber = TargetIdentifiers.TargetNumber
          AND Targets.Observer = TargetIdentifiers.Observer
          AND AircraftTargets.TargetNumber = Targets.TargetNumber
          AND AircraftTargets.Observer = Targets.Observer;
  INSERT INTO AirTargets VALUES $UniqueId, $AircraftType, $AircraftCategory, $Country;
  DELETE FROM Targets WHERE TargetNumber = $TargetNumber AND Observer =
  $Observer;
  DELETE FROM AircraftTargets
      WHERE TargetNumber = $TargetNumber AND Observer = $Observer;
END
```

One last rule uses predefined information about the suspicion code for an aircraft of a given category and from a given country to derive the suspicion codes for each aircraft target. The result is inserted into a table 'All Targets' that will also receive the results of processing ship targets. Note that a null value is inserted into the column that will hold the names of ships.

```
DEFINE CollateAirTargets AS
    SELECT * FROM AirTargets, AircraftSuspicion
```

```
        WHERE AirTargets.Country = AircraftSuspicion.Country
            AND AirTargets.AircraftCategory = AircraftSuspicion.AircraftCategory;
        INSERT INTO AllTargets
                VALUES $UniqueId, NULL, $AircraftType, $AircraftCategory,
                        $Country, $SuspicionCode;
        DELETE FROM AirTargets WHERE UniqueId = $UniqueId;
END
```

The rules for processing naval targets are more complicated, because the ship can be identified either by its name or by a combination of pennant number and country of origin. In addition, the ship category has to be derived once the ship's identity is unequivocally established.

```
DEFINE ProcessShipTargets AS
        SELECT TargetNumber, Observer, UniqueId, PennantNumber,
                ShipName, ShipClass, Country
        FROM Targets, NavalTargets, ShipClasses, TargetIdentifiers
        WHERE Targets.TargetType = 'Aicraft'
            AND Targets.TargetNumber = TargetIdentifiers.TargetNumber
            AND Targets.Observer = TargetIdentifiers.Observer
            AND NavalTargets.TargetNumber = Targets.TargetNumber
            AND NavalTargets.Observer = Targets.Observer;
        INSERT INTO ShipTargets
                VALUES $UniqueId, $PennantNumber, $ShipName, $ShipClass, $Country;
        DELETE FROM Targets WHERE TargetNumber = $TargetNumber
                AND Observer = $Observer;
        DELETE FROM ShipTargets WHERE TargetNumber = $TargetNumber
                AND Observer = $Observer;
END
```

When the ship name is provided, this is used to look up the details of the vessel in the 'Ships' and 'Ship Classes' tables. There is no need to explicitly require that the ship name is not null, because this causes the first condition in the clause to fail anyway. Also observe that the rest of the condition fails if the ship name does not appear in the 'Ships' table.

```
DEFINE ProcessNamedShips AS
        SELECT UniqueId, ShipName, Ships.ShipClass, Ships.Country, ShipCategory
        FROM ShipTargets, Ships, ShipClasses
        WHERE ShipTargets.ShipName = Ships.ShipName
            AND Ships.ShipClass = ShipClasses.ShipClass;
        INSERT INTO SeaTargets VALUES $UniqueId, $ShipName,
                $ShipClass, $ShipCategory, $Country;
```

```
        DELETE FROM ShipTargets WHERE UniqueId = $UniqueId;
    END
```

The rule below processes ship targets for which the ship's name is not given or does not appear in the 'Ships' table, but the pennant number and country of origin are provided and match an entry. Note that, as with the previous rule, the ship class and country from the predefined table are used in preference to the values provided in the incoming message.

```
    DEFINE ProcessUnnamedShips AS
        SELECT UniqueId, ShipName, Ships.ShipClass, Ships.Country, ShipCategory
        FROM ShipTargets, Ships, ShipClasses
        WHERE ShipTargets.ShipName NOT IN (SELECT ShipName FROM ShipTargets)
            AND ShipTargets.Country = Ships.Country
            AND ShipTargets.PennantNumber = Ships.PennantNumber;
            AND Ships.ShipClass = ShipClasses.ShipClass;
        INSERT INTO SeaTargets VALUES $UniqueId, $ShipName, $ShipClass,
        $ShipCategory, $Country;
        DELETE FROM ShipTargets WHERE UniqueId = $UniqueId;
    END
```

Any rows remaining in the 'ShipTargets' table after both of these rules are applied represent ships that cannot be properly identified. In this case, the ship class is used to try to determine the ship category.

```
    DEFINE UnidentifiedShip AS
        SELECT UniqueId, ShipClass, ShipCountry, ShipCategory
        FROM ShipTargets, ShipClasses
        WHERE ShipTargets.ShipClass = ShipClasses.ShipClass;
        INSERT INTO SeaTargets VALUES $UniqueId, NULL, $ShipClass, $ShipCategory,
        $Country;
        DELETE FROM ShipTargets WHERE UniqueId = $UniqueId;

    END
```

When the ship class cannot be determined, an error condition exists, because the output message format demands this information. One way of detecting this condition would be to construct a query that excludes all tuples satisfied by the where-clauses of the previous three rules, and then operates on the remainder. However, the query that would result is distinctly unwieldy; a better approach uses the suggested priority mechanism to examine the contents of the 'Ship Targets' table when the other rules are already applied.

```
    DEFINE ShipClassError :- ProcessNamedShips, ProcessUnnamedShips, UnidentifiedShip AS
        SELECT UniqueId FROM ShipTargets;
        WRITE ErrorLogFile 'Cannot determine ship class for target' $UniqueId '\n';
```

```
        DELETE FROM ShipTargets WHERE UniqueId = $UniqueId;
    END
```

The processed ship information can now be inserted into the table that is to contain all target information. At the same time, the predefined 'Ship Suspicion' table supplies the suspicion code for each ship target.

```
    DEFINE CollateSeaTargets AS
        SELECT * FROM SeaTargets, ShipSuspicion
        WHERE SeaTargets.Country = ShipSuspicion.Country
            AND SeaTargets.ShipCategory = ShipSuspicion.ShipCategory;
        INSERT INTO AllTargets VALUES $UniqueId, $ShipName, $ShipType, $ShipCategory,
            $Country, $SuspicionCode;
        DELETE FROM AirTargets WHERE UniqueId = $UniqueId;
    END
```

The last part of the input message format to be considered is the engagement information, which appears in the 'Engagements' table. The first thing to achieve is to change the combination of target number and observer into the unique identifier, and decompose the given date-time field into separate date and time values.

```
    DEFINE MapEngamentIdentifier AS
        SELECT TargetNumber, Observer, UniqueId, Date, Time, EngagementType, Outcome
        FROM Engagements, TargetIdentifiers, DatesAndTimes
        WHERE Engagements.TargetNumber = TargetIdentifiers.TargetNumber
            AND Engagements.Observer = TargetIdentifiers.Observer
            AND Engagements.Date-Time = DatesAndTimes.Date-Time;
        INSERT INTO Engagements1
                VALUES $UniqueId, $Date, $Time, $EngagementType, $Outcome;
        DELETE FROM Engagements
                WHERE TargetNumber = $TargetNumber AND Observer = $Observer;
    END
```

Now the table of all targets must be consulted for its suspicion code field. Using this, together with the predefined table 'Engagement Data', the required engagement description value can be found by performing yet another natural join.

```
    DEFINE DeriveEngagementDescription AS
        SELECT UniqueId, Date, Time, Description
        FROM Engagements1, EngagementData, AllTargets
        WHERE Engagements1.EngagementType = EngagementData.EngagementType
            AND Engagements1.Outcome = EngagementData.Outcome
            AND EngagementData.SuspicionCode = AllTargets.SuspicionCode
            AND AllTargets.UniqueId = Engagemements1.UniqueId;
```

```
        INSERT INTO EngagementDescriptions VALUES $UniqueId, $Date, $Time,
        $Description;
        DELETE FROM Engagements1 WHERE UniqueId = $UniqueId
    END
```

The tables 'Output Locations', 'All Targets', and 'Engagement Descriptions' contain the information that the composer can use to create output messages.

```
DEFINE SendLocationsTable AS
    SELECT * FROM OutputLocations;
    ACCEPT DummyComposer VALUES $UniqueId, $Date, $Time, $Latitude, $Longitude;
    DELETE FROM OutputLocations WHERE UniqueId = $UniqueId;
END
DEFINE SendAllTargets AS
    SELECT * FROM AllTargets;
    ACCEPT $UniqueId, $Name, $Type, $Category, $Country, $SuspicionCode;
    DELETE FROM AllTargets WHERE UniqueId = $UniqueId;
END
DEFINE SendDescriptions AS
    SELECT * FROM EngagementDescriptions AS
    ACCEPT $UniqueId, $Date, $Time, $Description;
    DELETE FROM EngagementDescriptions WHERE UniqueId = $UniqueId;
END
```

The composer is then responsible for interpreting these tables and building the appropriate output message from them.

References.

[1] Avo A.V., Ullman J. D., "Principles of Compiler Design", Addison-Wesley, Reading MA, 1977.

[2] Barrett W.A., Bates R.M., Gustafson D.A., Couch J.D., "Compiler Construction", 2nd Ed, Science Research Associates Inc, 1986.

[3] Batini C., Lenzerini M., Navathe S.B., "A Comparative Analysis of Methodologies for Database Schema Integration", ACM Computing Surveys Vol 18 No 4, 1986.

[4] Bouguettaya A., Papazoglou M., King R., "On Building a Hyperdistributed Database", Information Systems, Vol 20 No 7, pp557-577, 1995.

[5] Bowden F., "The ADF Joint Operations Simulation System", Proc SimTecT 97, pp441-446, 1997.

[6] Bowden F., Gabrisch G., Davies M., "The ADF Joint Operations Simulation of Air Defence C3I using the Distributed Interactive C3I Effectiveness (DICE) Simulation", Proc SimTecT 97, pp71-76, 1997.

[7] Breuer, P. T., Bowen J. P., "A PREttier Compiler-Compiler: Generating Higher-Order Parsers in C", Software - Practice and Experience, Vol 25 No 11, pp1263-1297, 1995.

[8] Bright M.W., Hurson A.R., Pakzad S., "Automated Resolution of Semantic Heterogeneity in Multidatabases", ACM Trans Database Systems, Vol 19 No 2, pp212-253, 1994.

[9] Brookes W., Induska J., Bond A., Yang Z., "Interoperability of Distributed Platforms: a Compatability Perspective", Proc 2nd Int Conf on Open Distributed Processing, pp67-78, 1995.

[10] Buvac S., Fikes R., "A Declarative Formalism of Knowledge Translation", Proc 1995 ACM CIKM Int Conf on Information and Knowledge Management, pp340-347, 1995.

[11] Cammarata S., Kameny I., Lender J., Replogle C., "A Metadata Management System to Support Data Interoperability, Reuse, and Sharing", Journal of Database Management, Vol 5 No 2, pp30-40, 1994.

[12] Cartwright D., "Building Bridges (Database Interoperability)", Network Week, Vol 2 No 10, pp49-52, 1996.

[13] Ceri S., Pelagatti G., "Distributed Databases: Principles and Systems", McGraw-Hill, NY, 1984.

[14] Colomb R.M., Orlowska M.E., "Interoperability in Information Systems", Information Systems Journal Vol 5 pp37-50, 1994.

[15] De Casto C., Grandi F., Scalas M.R., "Semantic Interoperability of Multitemporal Relational Databases", Proc 12th Int Conf on the Entity-Relationship Approach, pp463-474, 1993.

[16] Elmasri R., Navathe S., "Fundamentals of Database Systems", Benjamin/Cummings, Redwood City, Cal, 1989.

[17] Gelfond, M., Przymusinska, H., Przymusinski, T., "The Extended Closed World Assumption and its Relationship to Parallel Circumscription", Proc 5th SIGACT-SIGMOD Symp on Principles of Database Systems, pp133-139, 1986.

[18] Johnson, V. M., Carlis, J. V., "Building a Composite Syntax for Expert System Shells", IEEE Expert, pp60-66, 1997.

[19] Halpin T., "Conceptual Schema and Relational Database Design", 2nd Ed, Prentice-Hall, 1994.

[20] NASA Johnson Space Center, "NASA CLIPS Rule-Based Language", Available at http://www.siliconvalleyone.com/clips.htm.

[21] Ram S., Barkmeyer E, "A Unifying Semantic Model for Accessing Multiple Heterogeneous Databases in a Manufacturing Environment", Proc 1st Int Workshop on Interoperability in Multidatabase Systems, pp212-215, 1991.

[22] Reid D.J, Davies M., "Towards a Gateway to Interconnect Simulations and Operational C3I Systems", Proc SimTecT 98, pp27-32, 1998.

[23] Rusinkiewicz M.E., Sheth A.P., "Multidatabase Applications: Semantic and System Issues", Proc 18th Int Conf on Very Large Databases, 1992.

[24] Sciore H., Siegel M., Rosenthal A., "Using Semantic Values to Falicitate Interoperability among Heterogeneous Information Systems", ACM Trans on Database Systems, Vol 19 No 2, pp254-290, 1994.

[25] Takizawa M., Hasegawa M., Deen S., "Interoperability of Distributed Information Systems", Proc 1st Int Workshop on Interoperability in Multidatabase Systems, pp239-242, 1991.

[26] US Navy Center for Tactical Systems Interoperability, "Operational Specification for Over-the-Horizon Targeting Gold", Rev B Ch 2, 1996.

Translating Deeply Structured Information

Darryn J Reid

(DSTO-TR-0936)

DISTRIBUTION LIST

Number of Copies.

AUSTRALIA

DEFENCE ORGANISATION

Task sponsor:

DGC3ID	1

S&T Program

Chief Defence Scientist)	
FAS Science Policy)	1 shared copy
AS Science Corporate Management)	
Director General Science Policy Development	1
Counsellor, Defence Science, London	Doc Control Sheet
Counsellor, Defence Science, Washington	Doc Control Sheet
Scientific Adviser - Policy and Command	1
Navy Scientific Adviser	1 copy of Doc Control Sheet and 1 distribution list
Scientific Adviser - Army	1 copy of Doc Control Sheet and 1 distribution list
Air Force Scientific Adviser	1
Director Trials	1

Aeronautical & Maritime Research Laboratory

Director	1

Electronics and Surveillance Research Laboratory

Director	1 copy of Doc Control Sheet and 1 distribution list
Chief Information Technology Division	1
Research Leader Command & Control and Intelligence Systems	1
Research Leader Military Computing Systems	1
Research Leader Joint Systems Branch	1
Research Leader Advanced Computer Capabilities	Doc Control Sheet
Research Leader Command, Control and Communications	1
Head, Information Warfare Studies Group	Doc Control Sheet
Head, Software Systems Engineering Group	Doc Control Sheet
Head, Trusted Computer Systems Group	Doc Control Sheet
Head, Systems Simulation and Assessment Group	1
Head, C3I Operational Analysis Group	Doc Control Sheet

Head Information Management and Fusion Group	Doc Control Sheet
Head, Human Systems Integration Group	Doc Control Sheet
Head, C2 Australian Theatre	1
Head, Distributed Systems Group	Doc Control Sheet
Head C3I Systems Concepts Group	1
Head, Organisational Change Group	Doc Control Sheet
Task Manager	1
Author	2
Publications and Publicity Officer, ITD/ Executive Officer ITD	1

DSTO Library and Archives

Library Fishermens Bend	1
Library Maribyrnong	1
Library Salisbury	2
Australian Archives	1
Library, MOD, Pyrmont	Doc Control Sheet
US Defence Technical Information Center	2
UK Defence Research Information Centre	2
Canada Defence Scientific Information Service	1
NZ Defence Information Centre	1
National Library of Australia	1

Capability Systems Staff

Director General Maritime Development	Doc Control Sheet
Director General Aerospace Development	Doc Control Sheet

Army

ABCA Office, G-1-34, Russell Offices, Canberra	4
SO (Science), DJFHQ(L), MILPO, Enoggera, Qld 4051	Doc Control Sheet

Intelligence Program

DGSTA Defence Intelligence Organisation	1
Manager DIO Information Center	1

Corporate Support Program (libraries)

OIC TRS Defence Regional Library, Canberra	1

Universities and Colleges

Australian Defence Force Academy	
Library	1
Head of Aerospace and Mechanical Engineering	1
Deakin University, Serials Section (M list)), Deakin University Library,	1
Senior Librarian, Hargrave Library, Monash University	
Librarian, Flinders University	1

Other Organisations

NASA (Canberra)	1
AGPS	1
State Library of South Australia	1
Parliamentary Library, South Australia	1

OUTSIDE AUSTRALIA

Abstracting and Information Organisations	
Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
Materials Information, Cambridge Scientific Abstracts	1
Documents Librarian, The Center for Research Libraries, US	1
Information Exchange Agreement Partners	
Acquisitions Unit, Science Reference and Information Service, UK	1
Library - Exchange Desk, National Institute of Standards and Technology, US	1
SPARES	5
Total number of copies:	**56**

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

Translating Deeply Structured Information

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Darryn J Reid

5. CORPORATE AUTHOR

Electronics and Surveillance Research Laboratory
PO Box 1500
Salisbury SA 5108 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TR-0936	AR-011-201	Technical Report	February 2000

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
N9505/17/119	840786	DGC3ID	58	26

13. DOWNGRADING/DELIMITING INSTRUCTIONS

N/A

14. RELEASE AUTHORITY

Chief, Information Technology Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE CENTRE, DIS NETWORK OFFICE, DEPT OF DEFENCE, CAMPBELL PARK OFFICES, CANBERRA ACT 2600

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CASUAL ANNOUNCEMENT Yes

18. DEFTEST DESCRIPTORS

Command Control Communications and Intelligence
Relational Databases
Machine Translation

19. ABSTRACT

The problem of interfacing Command, Control, Communications and Intelligence (C3I) systems and applicable simulations is considered. In particular, this document focusses on the requirements and design of an engine to support translation between systems expecting to communicate using dissimilar message languages.

This engine interprets a given behavioural specification written in a high-level declarative language built around standard SQL. It is therefore natural and convenient to consider implementation using an appropriate relational database system to facilitate data storage and manipulation.

Beginning with an overview of the broader context and background, the discussion considers the way in which the semantic structures of the input and output languages can be captured using a conceptual modeling language. Such models can be readily mapped into a relational schema, and the actions that a translator should perform are easily expressed using SQL. Each of these actions must occur when a given set of conditions is satisfied; the engine is therefore a specialised rule-based system that manipulates the tuples of a relational database.